

# 软件测试基础与实践

**Software Testing: Foundations and Practices**

## 第2讲 白盒测试

教师：汪鹏 廖力

软件工程专业 主干课程



廖力

## 本讲内容

基本概念

白盒测试定义、特点、...

静态白盒测试

代码审查、桌面检查  
走查、静态测试工具

动态白盒测试

测试覆盖标准、基本路径测试、  
条件测试、数据流测试、循环测试；

白盒测试工具

主流白盒测试工具



廖力

## — 白盒测试概念

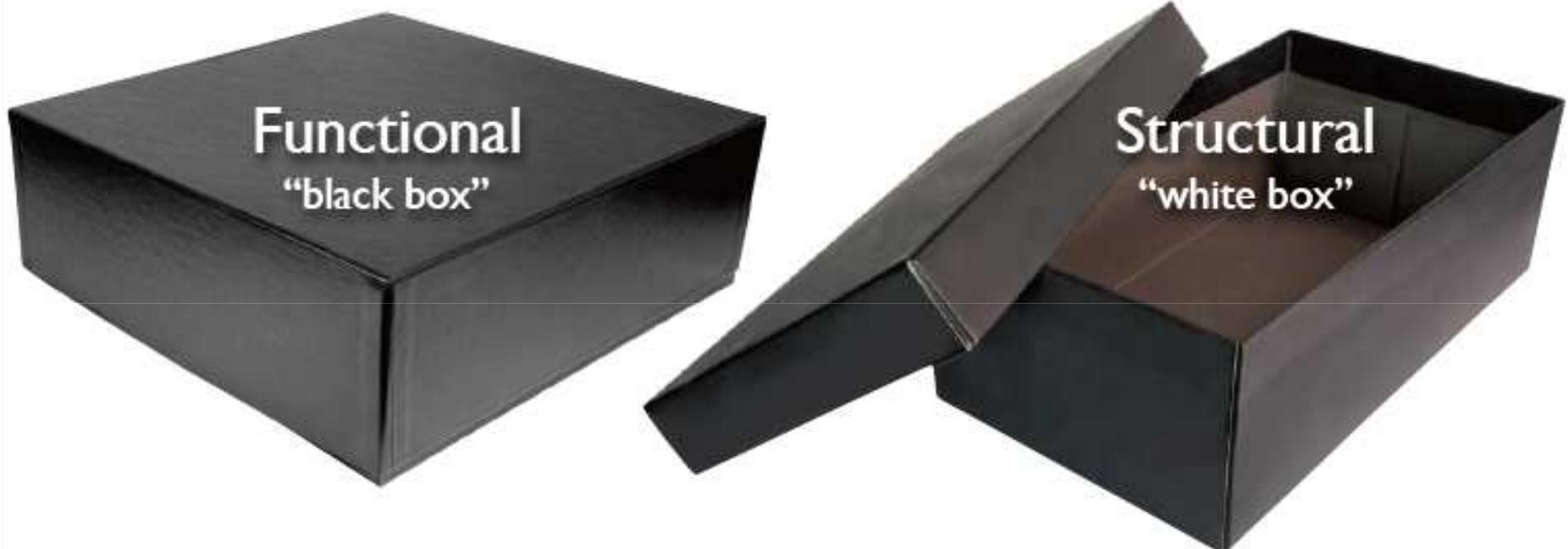


1. 为什么需要白盒测试？
2. 白盒测试有什么特点？
3. 如何实施白盒测试活动？



廖力

# Testing Tactics



- Tests based on spec
- Test covers as much *specified* behavior as possible

- Tests based on code
- Test covers as much *implemented* behavior as possible

# Why Structural?



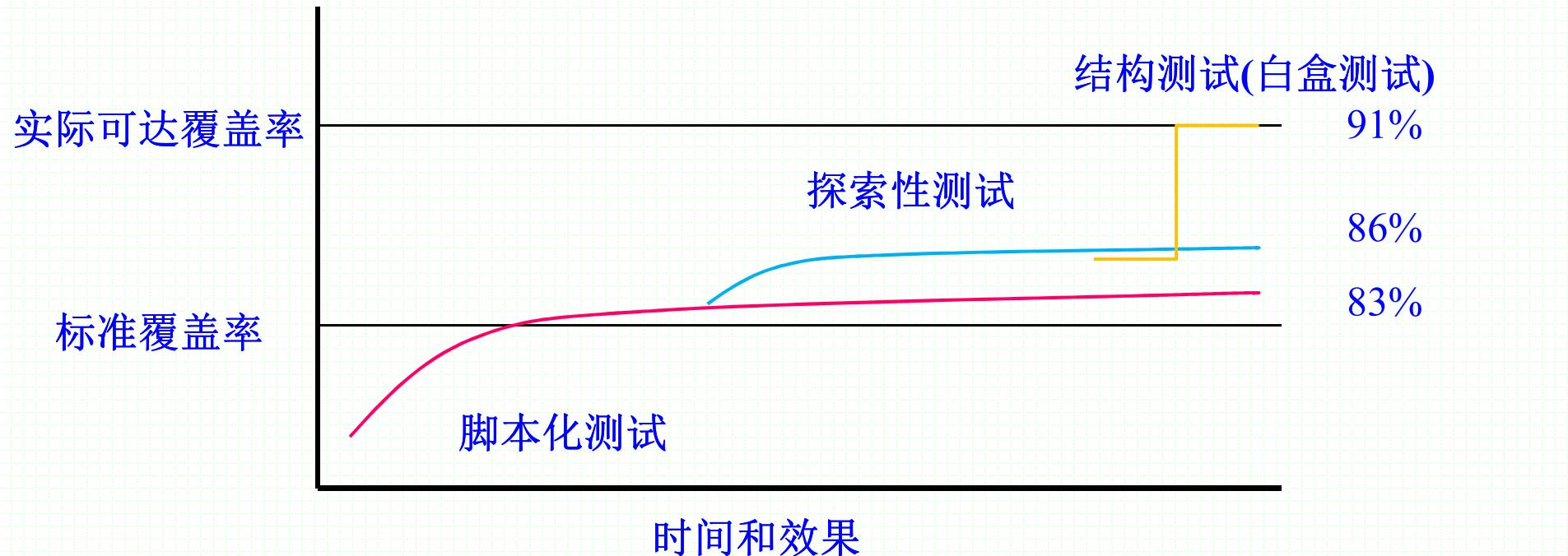
- If a part of the program is never executed, a defect may loom in that part  
A “part” can be a statement, function, transition, condition...
- Attractive because automated

# Why Structural?



- Complements functional tests  
Run functional tests first, then measure what is missing
- Can cover low-level details missed in high-level specification

探索性软件测试：一种测试人员在测试过程中利用获得的信息来直观地衍生出更多测试的测试执行方法——James Whittaker



廖力

## 白盒测试基本概念

### ■ 定义

白盒测试：一种基于源程序或代码的测试方法。依据源程序或代码结构与逻辑，生成测试用例以尽可能多地发现并修改源程序错误。

白盒测试分为静态和动态两种类型。

其它称谓：结构测试 逻辑驱动测试 基于程序的测试

### ■ 意义

主要的单元测试方法  
保证软件质量的基础



廖力

## 白盒测试基本概念

### ■ 实施者

单元测试阶段：一般由开发人员进行。

集成测试阶段：一般由测试人员和开发人员共同完成。

### ■ 步骤

#### 动态：

- (1) 程序图；
- (2) 生成测试用例；
- (3) 执行测试；
- (4) 分析覆盖标准；(5) 判定测试结果

#### 静态：

桌面检查；代码走查；代码审查



廖力

## 白盒测试基本概念

### ■ 进入和退出条件

进入条件：

编码开始阶段

退出条件：

- (1) 完成测试计划（满足一定覆盖率）
- (2) 发现并修正了错误
- (3) 预算和开发时间



廖力

## 二 静态白盒测试



1. 静态白盒测试有何特点？
2. 静态白盒测试具体方法是什么？
3. 静态白盒测试有哪些工具？



廖力

## 静态白盒测试

### ■ 定义

静态白盒测试：

在不执行软件的条件下有条理地仔细审查软件设计、体系机构和代码，从而找出软件缺陷的过程，有时称为结构化分析。

实施静态白盒测试的理由：

1. 尽早发现软件缺陷。
2. 为后继测试中设计测试用例提供思路。



廖力

## 实施静态白盒测试的理由：



- NASA软件工程实验室：阅读代码每小时能够检测出的缺陷要比测试高出80% (Basil and Selby 1987)
- Ackerman等1989：用测试来检测缺陷的成本是检查的6倍。
- IBM1995：检查发现一个错误需要3.5个工作时，测试需要花费15-25个小时。

——《代码大全2》中文版



廖力

## 静态白盒测试

### ■ 特点

测试条件：

只要求提供软件源代码，不要求提供可执行程序，即  
不用在计算机上执行程序，而是由人阅读代码。

测试目标：

- (1) 代码是否满足功能需求；
- (2) 代码是否与设计一致；
- (3) 代码是否遗漏某些功能；
- (4) 代码是否适当地处理错误



廖力

## 静态白盒测试

### ■ 优点

- (1) 可发现某些机器发现不了的错误
- (2) 利用不同人对代码的不同观点
- (3) 对照设计，确保程序能完成预期功能
- (4) 不但能检测出错误，还可以尝试确定错误根源
- (5) 节约计算机资源，但以增加人工成本为代价
- (6) 尽早发现缺陷，避免后期缺陷修复造成巨大压力



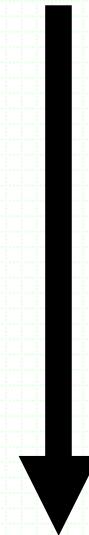
廖力

## 静态白盒测试

### ■ 静态白盒测试方法

- 1. 桌面检查 Desk Checking
- 2. 同行评审/检查/审查 Peer Review
- 3. 走查 Walkthrough
- 4. 代码评审/审查/检查 Inspection

形式化程度



廖力

## 静态白盒测试

### ■ 静态白盒测试方法——桌面检查

实施者：

通常是代码编写者

多数程序员在编译执行前都会做桌面检查

特点：

1. 无结构化或形式化方法保证
2. 不维护记录或检查单



廖力

## 静态白盒测试

### ■ 静态白盒测试方法——桌面检查

桌面检查记录格式：

- (1) 行号
- (2) 变量
- (3) 条件
- (4) 输入输出

输入：变量名+?+变量值

输出：变量名+=+变量值



廖 力

## 静态白盒测试

### ■ 静态白盒测试方法——桌面检查



1

```
1 calcWaterCost()
2 Input waterUsed
3 IF waterUsed<100 THEN
4   cost=50
5 ELSE
6   Input customer Type
7   IF customerType="D" THEN
8     cost=waterUsed*0.5
9   ELSE
10    cost=waterUsed*0.6
11   Display "Commerical rate"
12 ENDIF
13 Display "High usage"
14 ENDIF
15 Display cost
16 STOP
```



廖 力

## 静态白盒测试

### ■ 静态白盒测试方法——桌面检查

LN	cost	Customer type	water Used	Conditions	Input/Output
1					
2			200		waterUsed?200
3				200<100? Is F	
...	...	...	...	...	...
6		D			customerType?D
7				D=D? is T	
8	100				
...	...	...	...	...	...
13					High usage
14					
15					cost=100
16					



## 静态白盒测试

### ■ 静态白盒测试方法——桌面检查



```
1 calcSquares()  
2   Display "X", "X Squared"  
3   FOR x=1 TO 3 DO  
4     xSquared=x*x  
5     Display x,xSquared  
6   ENDFOR  
7 STOP
```

## 静态白盒测试

### 静态白盒测试方法——桌面检查

LN	x	xSquared	Conditions	Input/Output
1				
2				X, xSquared
3	1		1<=3?is T	
4		1*1=1		
5				X=1,xSquared=1
6	1+1=2			
3			2<=3?is T	
4		2*2=4		
5				X=2,xSquared=4
6	2+1=3			
...	...	...	...	...
7				



## 静态白盒测试

### ■ 静态白盒测试方法——桌面检查

优点：

1. 编码者容易理解和阅读自己代码
2. 开销小，没有指定进度
3. 尽早发现缺陷

缺点：

1. 开发人员不是实施测试的最佳人选
2. 依靠个人勤奋和技能，有效性难以保证



廖力

## 静态白盒测试

### ■ 静态白盒测试方法——Peer Review

召集小组成员进行的初次审查。2-3人聚集起来审查代码，寻找问题和失误。

特点：

1. 团体小，仅有开发者和充当审查者的其他一两个程序员和测试员。
2. 非正式。



廖力

## 静态白盒测试

■ 静态白盒测试方法——**代码走查 Walkthrough**  
比Peer Review更为正规化的下一步。以组为单  
位进行代码阅读的测试方法

特点：

1. 由特定人员组成的团队通过会议完成
2. 与会者充当计算机执行测试用例
3. 开发人员及时回答与会者提出的问题



廖力

## 静态白盒测试

### 成员构成：

主持人、记录员、编码者、测试人员、其他项目成员。

审查人员中至少有一位资深程序员。

### 走查规程：

1. 审查员在审查前接到软件拷贝，事先准备
2. 作者逐行或者逐个功能通读代码，解释代码
3. 审查人员聆听，充当“计算机”模拟运行，提出有疑义的问题
4. 审查后，作者编写报告，说明发现了什么问题，计划如何解决。



廖力

## Part Review

### ■ 静态白盒测试方法

- 1. 桌面检查 Desk Checking
- 2. 同行评审/检查/审查 Peer Review
- 3. 走查 Walkthrough
- 4. 代码评审/审查/检查 Inspection

形式化程度

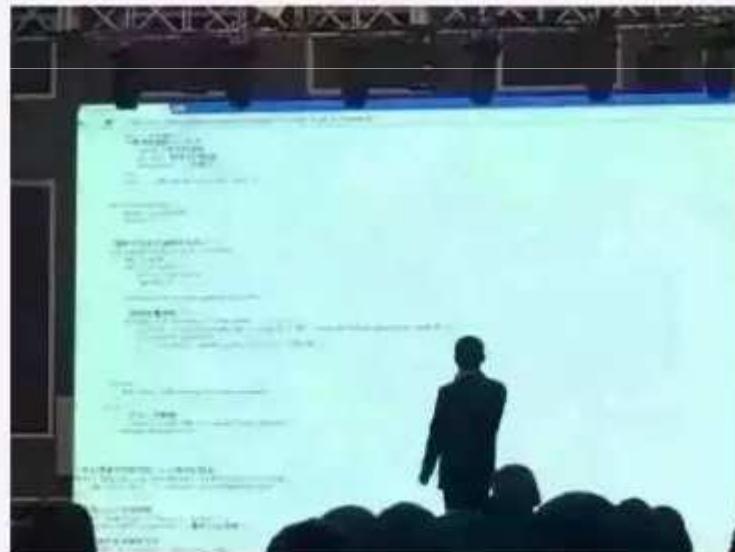


软

Software



公司年会，中奖 😍 概率有点“耐人寻味”，CTO说要回去review抽奖程序的代码 😳，于是倒霉的程序员自己主动上台展示源代码 😱，出现了大会现场一千多名的研发同事review抽奖代码的镜头 😳，场面有点壮观 😱 求写此代码程序猿的心理阴影面  
全文



19分钟前



廖力

2019年9月25日

## 静态白盒测试

### ■ 静态白盒测试方法——代码审查 Inspection

- 1976年 Fagan
- 1993年 Gilb and Graham
- 1996年 最佳软件实践



Michael Fagan

代码审查是有力的质量技术

*Fagan, M. "Design and Code Inspections to Reduce Errors in Program Development."  
IBM Systems Journal 15, 3 (1976): 182-211.*



廖力

## 静态白盒测试

### ■ 静态白盒测试方法——代码审查

Inspection是最正式的审查类型，具有高度组织化，要求每个参与者都接受训练。

#### 代码审查小组：

- (1) 主持人：多面手
- (2) 表述者：会上通读程序，但不是作者
- (3) 作者：解释
- (3) 评论员：找出缺陷
- (4) 记录员：



廖力

## 静态白盒测试

### ■ 静态白盒测试方法——代码审查

代码审查步骤：

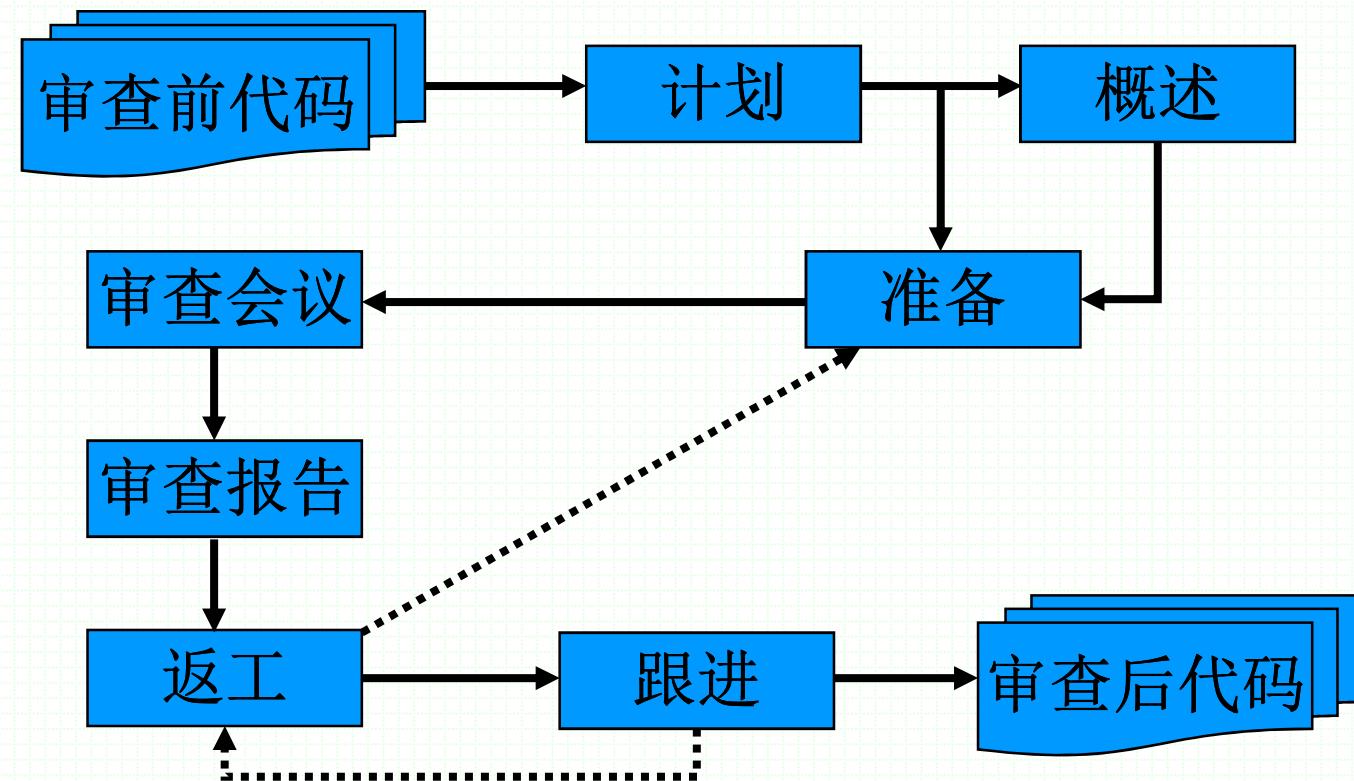
- (1) 计划(Plan)
- (2) 概述(Overview Meeting)
- (3) 准备(Preparation)
- (4) 审查会议(Inspection Meeting)
- (5) 审查报告(Report)
- (6) 返工(Rework)
- (7) 跟进(Follow up)



廖力

## 静态白盒测试

### ■ 静态白盒测试方法——代码审查



廖力

## 静态白盒测试

### ■ 静态白盒测试方法——代码审查

代码审查步骤：

(1) 计划(Plan)

    主持人做计划

(2) 概述(Overview Meeting)

    描述技术背景

(3) 准备(Preparation)

    评论员审查代码      核对表



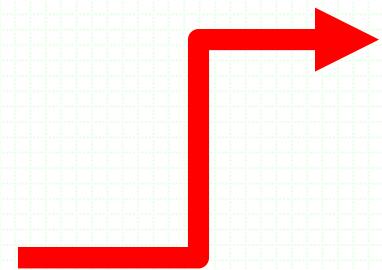
廖力

## 静态白盒测试

### ■ 静态白盒测试方法——代码审查

核对表例1：

- 1. 数据引用错误
- 2. 数据声明错误
- 3. 计算错误
- 4. 比较错误
- 5. 控制流错误
- 6. 接口错误
- 7. 输入/输出错误



- 5. 控制流错误
  - 5. 1循环是否能中止
  - 5. 2程序、模块和子程序是否会中止
  - 5. 3某些取值是否会导致循环从不被执行
  - .....



廖力

## 静态白盒测试

### ■ 静态白盒测试方法——代码审查

代码审查步骤：

(4) 审查会议 (Inspection Meeting)

阅读代码      讨论      提问

记录错误：类型、严重级别

讨论速度

不讨论解决方案

(5) 审查报告 (Report)

缺陷列表：类型、严重级别

核对表的基础



廖力

## 静态白盒测试

### ■ 静态白盒测试方法——代码审查

代码审查步骤：

(6) 返工 (Rework)

分配缺陷并修复

(7) 跟进 (Follow up)

监督，审查修复部分



廖力

## 静态白盒测试

### ■ 静态白盒测试方法——代码审查

代码审查作用：

- (1) 发现代码缺陷
- (2) 提高代码质量
- (3) 及早定位缺陷群集位置



廖力

## 静态白盒测试

### ■ 静态白盒测试方法——代码审查

代码审查挑战：

- (1) 很费时间
- (2) 涉及多人参加（支持条件 进度安排）
- (3) 不能保证参与者全部理解程序



廖 力

## 静态白盒测试

### ■ 静态白盒测试工具

代码静态分析工具（编译器的扩展）

作用：

- (1) 是否有不可达代码
- (2) 定义变量没有使用
- (3) 分配内存但没有释放
- (4) 计算圈复杂度



廖力

## 不可达代码

```
int f (int x, int y)
{
    return x+y;
    int z = x*y;
}
```

```
int n = 2+1;
if (4==n)
{
    xyz();
}
```

```
double x = sqrt(2);
if (x>5)
{
    xyz();
}
```



廖 力

## *More than 100 code analysis tools...*

[http://en.wikipedia.org/wiki/List\\_of\\_tools\\_for\\_static\\_code\\_analysis](http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis)

### **.NET (C#, VB.NET and all .NET compatible languages)**

- [FxCop](#) — Free static analysis for Microsoft .NET programs that compile to [CIL](#). Standalone and [Visual Studio](#) editions. From Microsoft.
- [StyleCop](#) — Analyzes C# source code to enforce a set of style and consistency rules. It can be run in [Visual Studio](#) or integrated into an [MSBuild](#) project. Free download from Microsoft.

### **Java**

- [Checkstyle](#) — besides some static code analysis, it can be used to show violations of a configuration file.
- [FindBugs](#) — an open-source static bytecode analyzer for Java (based on [Jakarta BCEL](#)) from the Eclipse Foundation.
- [PMD \(software\)](#) — a static ruleset based Java source code analyzer that identifies potential problems.
- [Hammurapi](#) — a versatile code review solution.
- [Sonar](#) — a platform to manage source code quality
- [Soot](#) — a language manipulation and optimization framework consisting of intermediate language
- [Squale](#) — a platform to manage software quality (also available for other languages, using common interfaces).

### **C**

- [Sparse](#) — A tool designed to find faults in the [Linux](#) kernel.
- [Splint](#) — An open source evolved version of Lint (C language).
- [Uno](#) — A tool designed to find most common type of programming errors without generating too many false positives.
- [BLAST](#) (Berkeley Lazy Abstraction Software verification Tool) — a software model checker for C abstraction.
- [Frama-C](#) — A static analysis framework for C.

## 静态白盒测试

### ■ 静态白盒测试工具举例

#### 代码评审辅助工具

作用：

- (1) 提供良好的协同评审环境
- (2) 提高评审效率
- (3) 具有一定的代码自动分析功能



廖力

## *More than 100 code inspection tools...*

[http://www.laatuk.com/tools/review\\_tools.html](http://www.laatuk.com/tools/review_tools.html)

<u>DevInspect</u>	Windows	<u>HP</u>	Static and dynamic analysis of security vulnerabilities
<u>AppScan</u>	Windows	<u>IBM</u>	Static and dynamic application security testing suite that scans and tests for all common web application vulnerabilities
<u>Veracode</u>	SaaS	<u>Veracode, Inc.</u>	On-demand service which analyzes both static binaries and running web-based applications for security flaws
<u>Ounce</u>	AIX, Linux, Windows, Solaris	<u>Ounce Labs, Inc.</u>	Analysis security vulnerabilities from source code
<u>Fortify 360: Source Code Analyzer</u>	Windows	<u>Fortify Software, Inc.</u>	Analysis security vulnerabilities from source code
<u>Google Code's code review tool</u>	web	<u>Google Code</u>	review tool for Google Code
<u>Crucible</u>	Java	<u>Atlassian</u>	A tool for running code reviews (review changes, make comments, manage review workflow, etc.)
<u>FishEye</u>	Java	<u>Atlassian</u>	A tool for investigating changes made in source code repository
<u>Checkstyle</u>		<u><a href="http://checkstyle.sourceforge.net/">http://checkstyle.sourceforge.net/</a></u>	
<u>FindBugs</u>		<u><a href="http://findbugs.sourceforge.net/">http://findbugs.sourceforge.net/</a></u>	Static analysis tool for Java code

## 三 动态白盒测试



1. 动态白盒简介
2. 逻辑覆盖
3. 基本路径方法
4. 循环测试方法
5. 数据流覆盖



廖力

## 动态白盒测试

### ■ 特点

- 不但要提供软件源代码，还要提供可执行程序，测试过程需要在计算机上执行程序。

### ■ 动态白盒测试试图检查

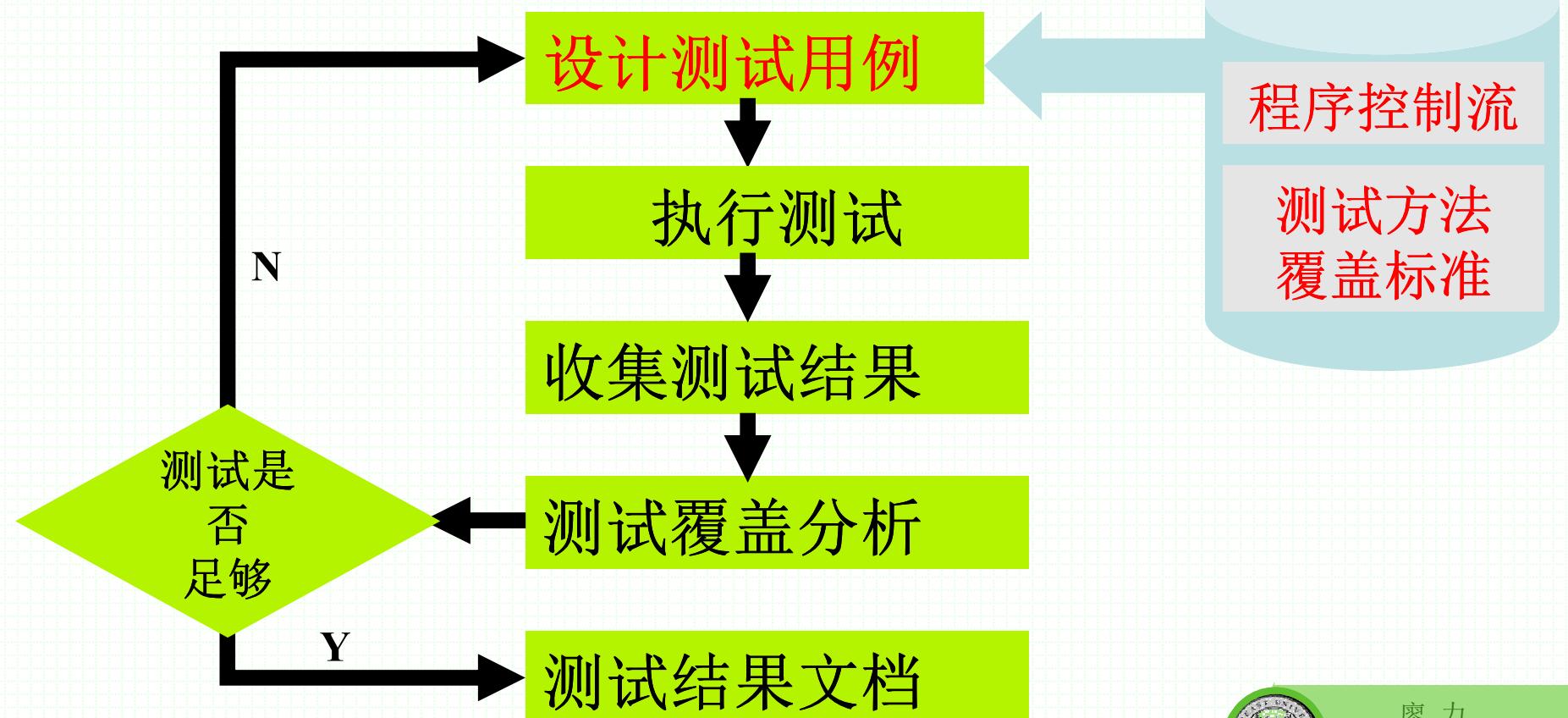
- 对程序模块中的所有独立执行路径至少执行一次
- 对所有逻辑判定的取值（“真”与“假”）都至少测试一次
- 在上下边界及可操作范围内运行所有循环
- 测试内部数据结构的有效性
- 等等



廖力

## 动态白盒测试——流程

### ■ 动态白盒测试流程



## 动态白盒测试——覆盖准则简介

### ■ 意义：

对“测试执行到何时才是足够？”的定量回答

### ■ 作用：

测试软件的一种度量标准，描述程序源码被测试的程度

### ■ 一种测试技术通常有一种对应的覆盖准则



廖力

## 动态白盒测试——覆盖准则简介

### ■ 动态白盒测试方法

#### 基于控制流的方法

- 1. 语句覆盖测试——语句覆盖
- 2. 条件测试——判定覆盖、条件覆盖
- 3. 路径测试——路径覆盖

#### 基于数据流的方法

- 4. 数据流测试



廖力

## 动态白盒测试——覆盖准则简介

### ■ 逻辑覆盖

白盒测试要求对被测程序结构特性做到一定程度的覆盖

#### 常用覆盖准则

1	语句覆盖	保证程序中的每条语句都执行一遍
2	判定覆盖	保证每个判断取True和False至少一次
3	条件覆盖	保证每个判断中的每个条件的取值至少满足一次
4	判定条件覆盖	保证每个条件和由条件组成的判断的取值...
5	条件组合覆盖	保证每个条件的取值组合至少出现一次...
6	路径覆盖	覆盖程序中所有可能路径



廖力



对下列子程序进行测试

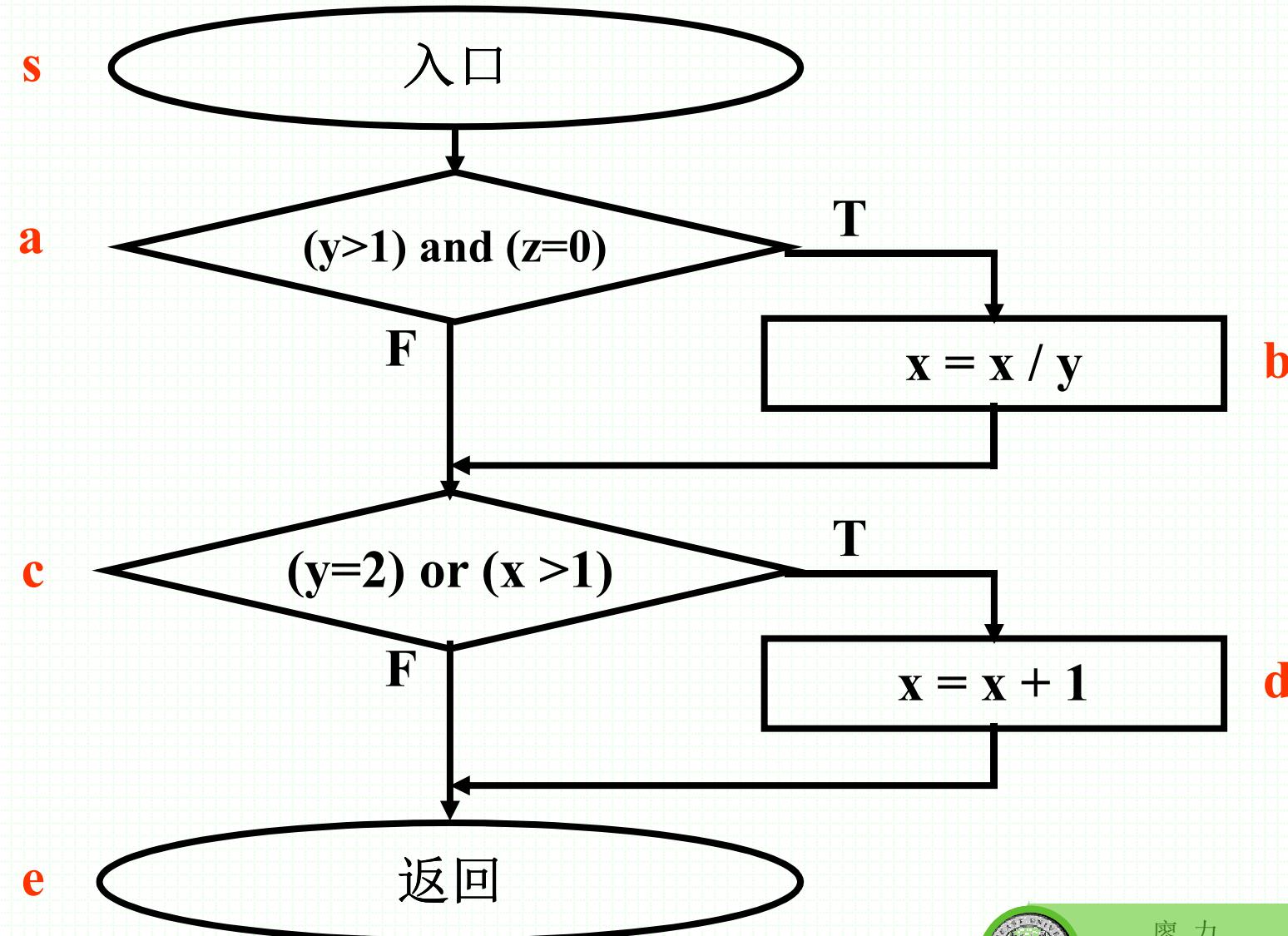
```
procedure example(y,z:real;var x:real);
begin
    if (y>1) and (z=0) then x:=x/y;
    if (y=2) or (x>1) then x:=x+1;
end;
```

该子程序接受x、y、z的值，并将计算结果x的值返回给调用程序。

与该子程序对应的流程图如下：



廖力



## 动态白盒测试——语句覆盖

■思想：语句覆盖是指选择足够的测试用例，使得运行这些测试用例时，被测程序的每个可执行语句都至少执行一次

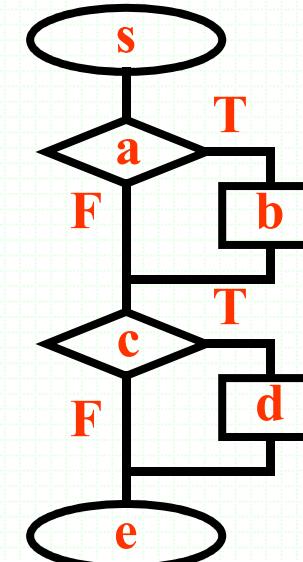
语句覆盖(**Statement Coverage**)其它称谓：

线覆盖面(Line Coverage) 段覆盖面(Segment Coverage)



廖力

- 欲使每个语句都执行一次，只需执行路径  $sabcde$  即可。
- 测试用例  $(y, z, x)$ :
  - 输入  $(2, 0, 4)$ ，输出  $(2, 0, 3)$
- 是否能发现遗漏边



a:  $(y > 1) \text{ and } (z = 0)$

c:  $(y = 2) \text{ or } (x > 1)$



廖 力

## 控制流覆盖

### ■ 语句覆盖

100%的语句覆盖可能很困难

- (1) 处理错误的代码片段
- (2) 小概率事件(软件开发中的恶作剧)
- (3) 不可达代码

```
if (x>y && y>z && z>x){  
    cout<<“This should never happen!”;  
}
```



廖力

## 控制流覆盖

### ■ 语句覆盖

脆弱的100%语句覆盖

无法发现某些严重问题

```
int* p=NULL;  
if (condition) {  
    p=&variable;  
}  
*p=123;
```

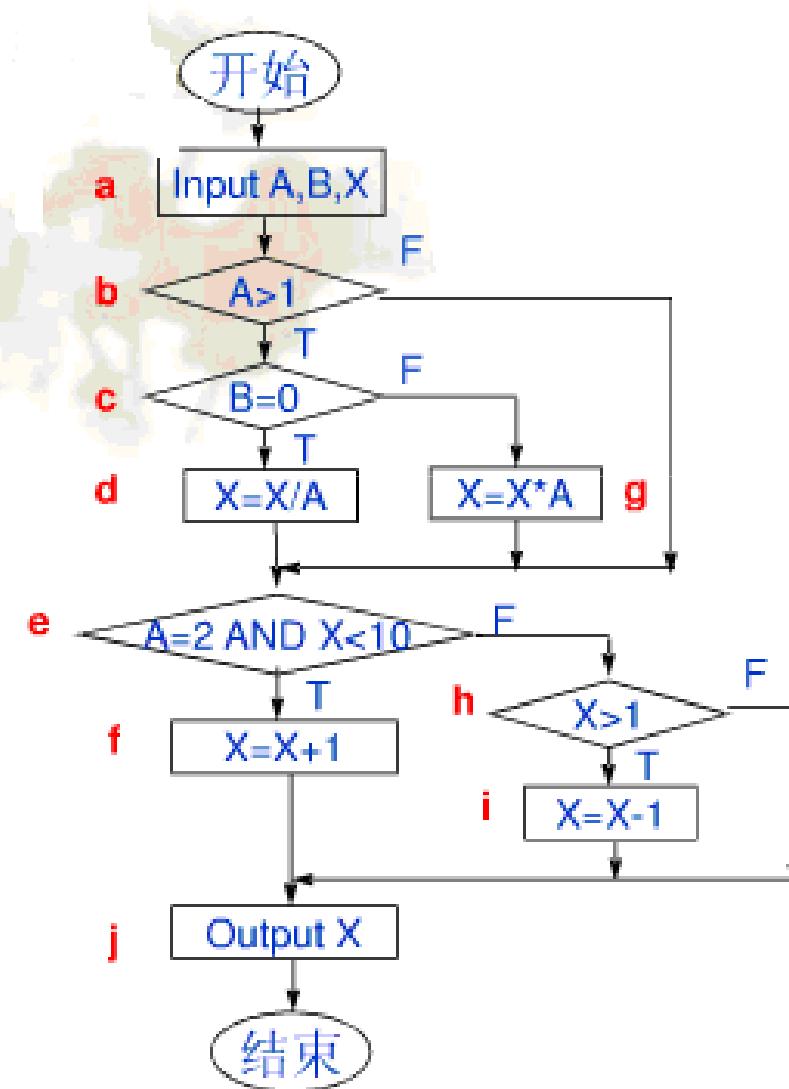


廖力

## 练习

\*\*

```
START  
INPUT (A, B, X)  
IF A>1  
    THEN IF B=0  
        THEN X:=X/A  
        ELSE X:=X*A  
    ENDIF  
ENDIF  
IF (A=2 and X<10)  
    THEN X=X+1  
    ELSE IF X>1  
        THEN X=X-1  
    ENDIF  
ENDIF  
OUTPUT (X)  
STOP
```



- 动态白盒覆盖标准回顾

常用逻辑覆盖准则		
1	语句覆盖	保证程序中的 <b>每条语句</b> 都执行一遍
2	判定覆盖	保证 <b>每个判断</b> 取True和False至少一次
3	条件覆盖	保证每个判断中的每个条件的取值至少满足一次
4	判定条件覆盖	保证每个条件和由条件组成的判断的取值...
5	条件组合覆盖	保证每个条件的取值组合至少出现一次...
6	路径覆盖	覆盖程序中所有可能路径



廖力

## 动态白盒测试——判定覆盖

思想：判定覆盖（也称**分支覆盖**）是指选择足够的测试用例，使得运行这些测试用例时，被测程序的每个判定的所有可能结果都至少执行一次（即判定的每个分支至少经过一次）



## 控制流覆盖

### ■ 判定覆盖

判定覆盖(Decision Coverage):

保证每个判断至少获得一次True和False

其它称谓：

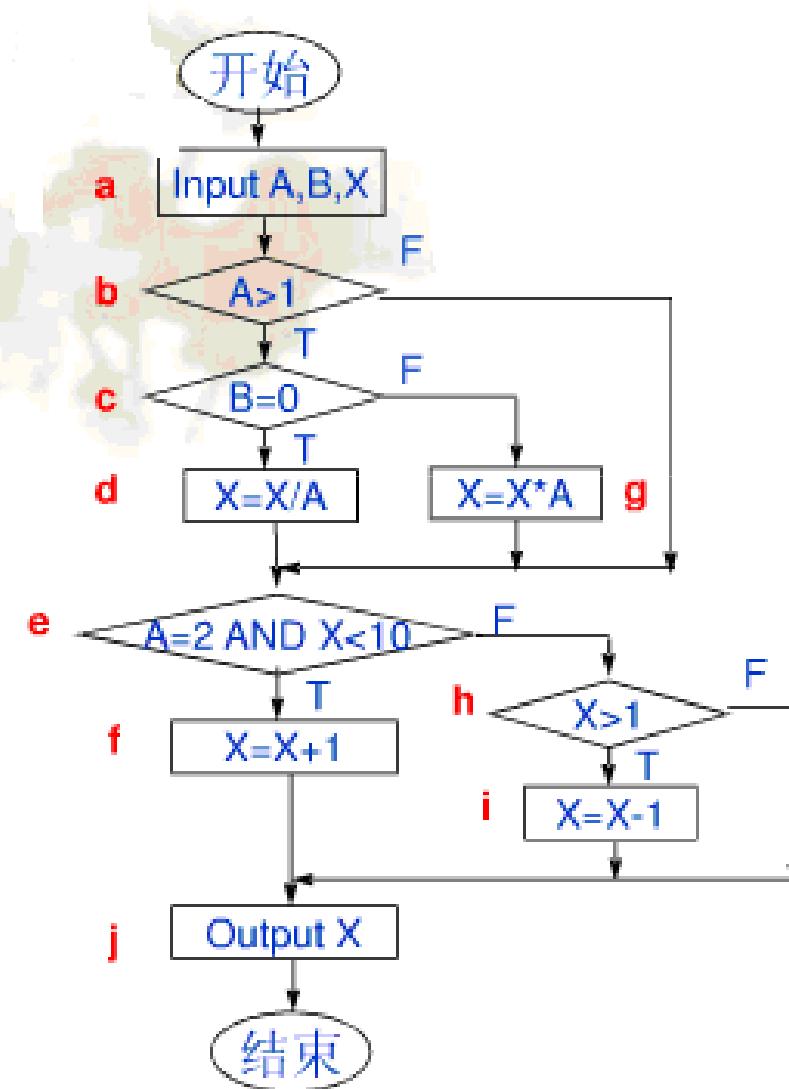
分支覆盖 所有边覆盖



廖力

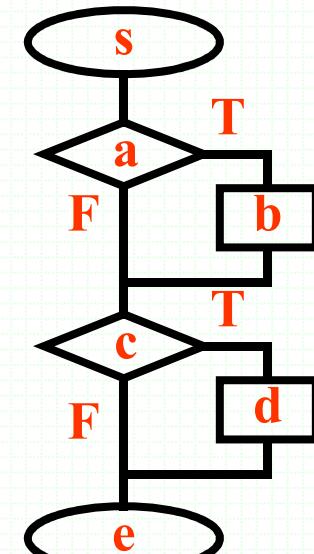
## 练习

```
START  
INPUT (A, B, X)  
IF A>1  
    THEN IF B=0  
        THEN X:=X/A  
        ELSE X:=X*A  
    ENDIF  
ENDIF  
IF (A=2 and X<10)  
    THEN X=X+1  
    ELSE IF X>1  
        THEN X=X-1  
    ENDIF  
ENDIF  
OUTPUT (X)  
STOP
```



- 保证每个判断取真假至少一次  
(即判定的每个分支至少经过一次)
  - 路径:  $sacde+sabce; \quad sabcde+sace$
  - 测试用例  $(y, z, x)$ :

测试数据	预期结果	路径	a	c
2, 1, 1	2, 1, 2	sacde	f	t
3, 0, 3	3, 0, 1	sabce	t	f



**a:**  $(y > 1) \text{ and } (z = 0)$   
**c:**  $(y = 2) \text{ or } (x > 1)$



## 控制流覆盖

### ■ 判定覆盖

例子：控制结构的执行不依赖条件function1()

```
if (condition1 && (condition1 || function1()))  
    statement1;  
  
else  
    statement2;
```



廖力

## 控制流覆盖

### ■ 判定覆盖

优点：

简单，**包含语句覆盖并避免了语句覆盖的问题，**

缺点：

忽略了表达式内的条件，不能发现每个条件的错误



廖力

## 动态白盒测试——条件覆盖

思想：保证每个判断中的每个条件的取值至少满足一次

例： $x > 3 \text{ OR } y < 5$

两个原子条件： $x > 3$      $y < 5$

测试用例1： $x=4 \ y=8$       true OR false = true

测试用例2： $x=3 \ y=3$       false OR true = true



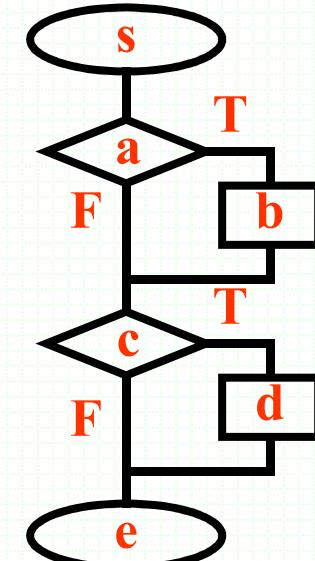
廖力

- 选择足够的测试用例，使得运行这些测试用例时，被测程序的每个判定中的每个条件的所有可能结果都至少出现一次)

- 测试用例  $(y, z, x)$ :

条件覆盖不能保证程序所有分支都被执行

测试数据	预期结果	路径	覆盖条件	a	c
1, 0, 3	1, 0, 4	sacde	F T F T	F	T
2, 1, 1	2, 1, 2	sacde	T F T F	F	T



**a:**  $(y > 1) \text{ and } (z = 0)$   
**c:**  $(y = 2) \text{ or } (x > 1)$



## 动态白盒测试——条件覆盖

思想：保证每个判断中的每个条件的取值至少满足一次

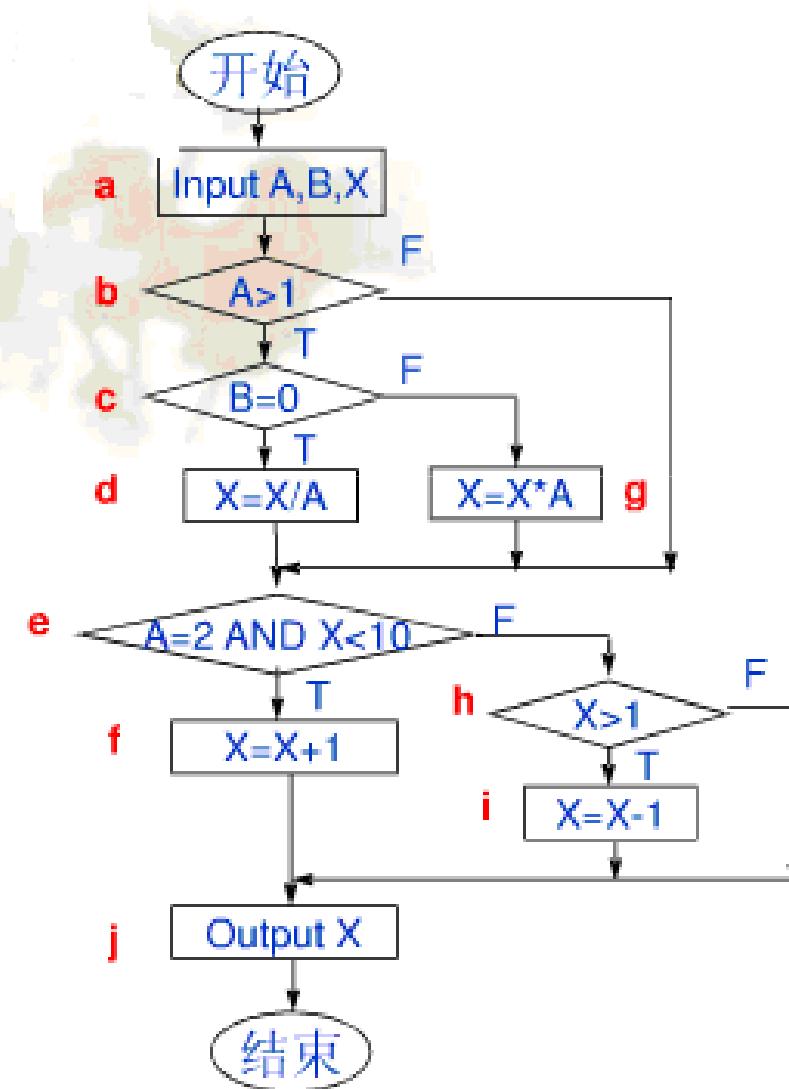
条件覆盖不要求覆盖判定结果的不同逻辑值，因此可能比语句覆盖或判定覆盖要弱



廖力

#### 练习

```
START  
INPUT (A, B, X)  
IF A>1  
    THEN IF B=0  
        THEN X:=X/A  
        ELSE X:=X*A  
    ENDIF  
ENDIF  
IF (A=2 and X<10)  
    THEN X=X+1  
    ELSE IF X>1  
        THEN X=X-1  
    ENDIF  
ENDIF  
OUTPUT (X)  
STOP
```



## 动态白盒测试——判定条件覆盖

思想：保证每个条件和由条件组成的判断的取值

例： $x > 3 \text{ OR } y < 5$

(1)  $x=6 \ y=3 \quad T \text{ OR } T = T \quad (\text{可省略})$

(2)  $x=6 \ y=8 \quad T \text{ OR } F = T \quad (\text{保留})$

(3)  $x=2 \ y=3 \quad F \text{ OR } T = T \quad (\text{保留})$

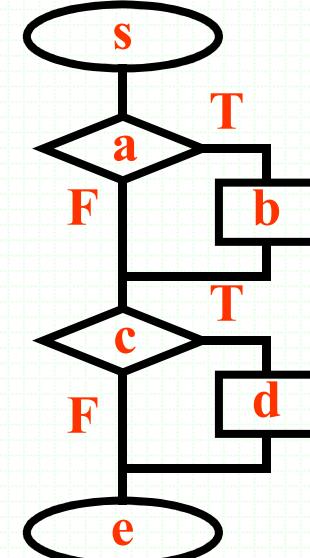
(4)  $x=2 \ y=8 \quad F \text{ OR } F = F \quad (\text{保留})$



廖力

- 被测程序的每个判定的所有可能结果都至少执行一次，并且，每个判定中的每个条件的所有可能结果都至少出现一次
- 测试用例  $(y, z, x)$ :

测试数据	预期结果	路径	覆盖条件	a	c
1, 0, 3	1, 0, 4	sacde	F / F T	F	T
2, 1, 1	2, 1, 2	sacde	T F T /	F	T
3, 0, 0	3, 0, 1	sabce	T T F F	T	F

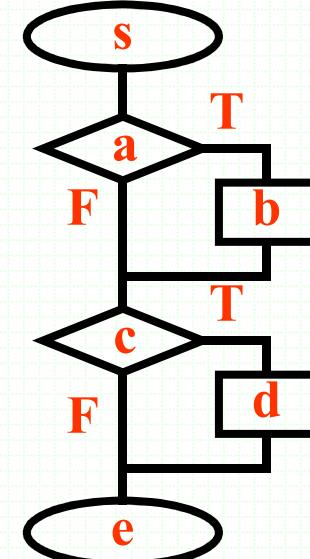


**a:**  $(y > 1) \&& (z = 0)$   
**c:**  $(y = 2) \parallel (x > 1)$



- 被测程序的每个判定的所有可能结果都至少执行一次，并且，每个判定中的每个条件的所有可能结果都至少出现一次
- 测试用例  $(y, z, x)$ :

测试数据	预期结果	路径	覆盖条件	a	c
1, 0, 3	1, 0, 4	sacde	F T F T	F	T
2, 1, 1	2, 1, 2	sacde	T F T F	F	T
3, 0, 0	3, 0, 1	sabce	T T F F	T	F

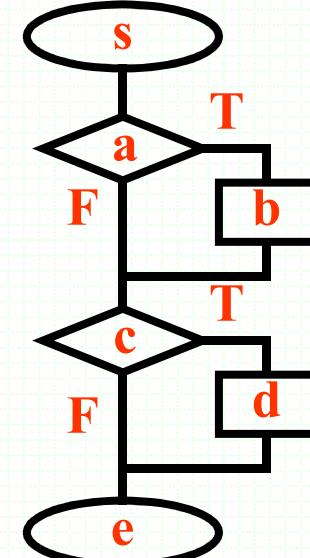


**a:**  $(y > 1) \text{ and } (z = 0)$   
**c:**  $(y = 2) \text{ or } (x > 1)$



- 被测程序的每个判定的所有可能结果都至少执行一次，并且，每个判定中的每个条件的所有可能结果都至少出现一次
- 测试用例  $(y, z, x)$ :

测试数据	预期结果	路径	覆盖条件	a	c
2, 0, 3	-	sabcde	T T T T	T	T
1, 1, 1	-	sace	F F F F	F	F



**a:**  $(y > 1) \text{ and } (z = 0)$   
**c:**  $(y = 2) \text{ or } (x > 1)$



## 动态白盒测试——判定条件覆盖

- 对于条件的每个组合，首先必须确定哪个测试用例对发现故障是敏感的，哪个组合产生的故障会被屏蔽。造成故障被屏蔽的组合在测试过程中可以不考虑，因此需要考虑的测试用例数量要少一些。
- **错误屏蔽：**指原子条件取值改变不会影响判定结果，因此该条件上的取值错误是不可见的。



廖力

## 动态白盒测试——判定条件覆盖

ID	原子条件			判定
	a	b	c	
1	T	T	T	T
2	F	F	F	F

100% 覆盖

ID	原子条件			判定	
	a	b	c	$a \&& (b \parallel c)$	$a \&& (b \& \& c)$
1	T	T	T	T	T
2	F	F	F	F	F

100% 覆盖但无法  
检测出运算符错误



廖力

## 动态白盒测试——判定条件覆盖

ID	原子条件			判定
	a	b	c	
1	T	F	T	T
2	F	T	F	F

100%覆盖

ID	原子条件			判定	
	a	b	c	$a  (b  c)$	$a&&(b&&c)$
1	T	F	T	T	F
2	F	T	F	T	F

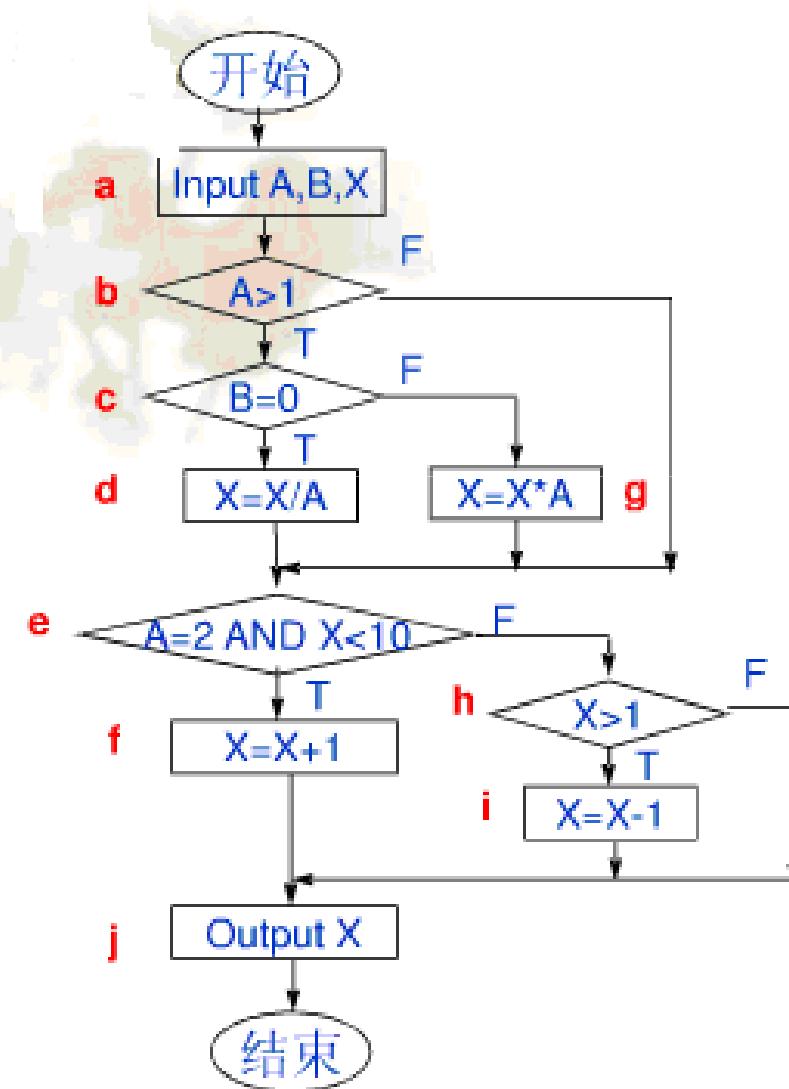
100%覆盖,检测出  
运算符错误



廖力

## 练习

```
START  
INPUT (A, B, X)  
IF A>1  
    THEN IF B=0  
        THEN X:=X/A  
        ELSE X:=X*A  
    ENDIF  
ENDIF  
IF (A=2 and X<10)  
    THEN X=X+1  
    ELSE IF X>1  
        THEN X=X-1  
    ENDIF  
ENDIF  
OUTPUT (X)  
STOP
```



## 动态白盒测试——条件组合覆盖

思想：保证每个条件的取值组合至少出现一次

例： $x > 3 \text{ OR } y < 5$

(1)  $x=6 \ y=3 \quad T \text{ OR } T = T$

(2)  $x=6 \ y=8 \quad T \text{ OR } F = T$

(3)  $x=2 \ y=3 \quad F \text{ OR } T = T$

(4)  $x=2 \ y=8 \quad F \text{ OR } F = F$

代价昂贵： $2^n$ 组合数目， $n$ 为原子条件数



廖力

## 动态白盒测试——条件组合覆盖

例:  $x >= 3 \text{ AND } x < 5$

- (1)  $x=4$ : **T and T = T**
- (2)  $x=8$       **T and F = F**
- (3)  $x=1$       **F and T = F**
- (4)  $x=?$       **F and F = F**

某些条件组合是不可能的



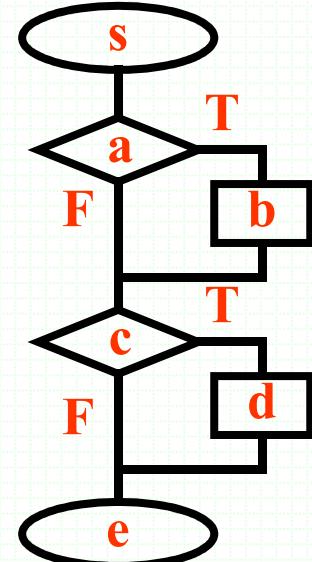
廖力

- 测试用例  $(y, z, x)$  (不考虑短路)

•

测试数据	预期结果	路径	覆盖的条件
2, 0, 4	2, 0, 3	sabcde	T T T T
2, 1, 1	2, 1, 2	sacde	T F T F
1, 0, 3	1, 0, 4	sacde	F T F T
1, 1, 1	1, 1, 1	sace	F F F F

是否经过sabce路径?



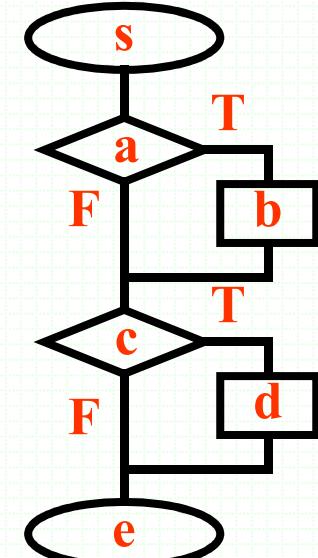
a:  $(y > 1) \text{ and } (z = 0)$   
c:  $(y = 2) \text{ or } (x > 1)$



廖力

- 测试用例  $(y, z, x)$  (考虑短路) :

测试数据	预期结果	路径	覆盖的条件
2, 0, 4	2, 0, 3	sabcde	T T T /
3, 1, 3	3, 1, 4	sacde	T F F T
1, 1, 1	1, 1, 1	sace	F / F F



a:  $(y > 1) \&& (z = 0)$   
c:  $(y = 2) \parallel (x > 1)$

是否还有遗漏路径?



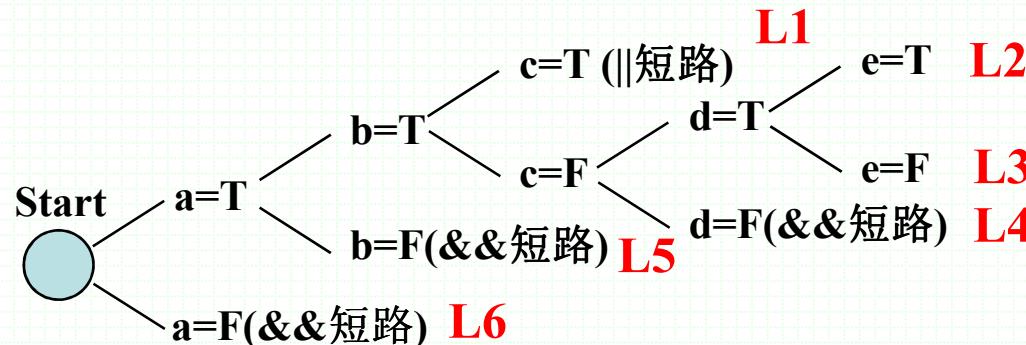
廖力

## 动态白盒测试——条件组合覆盖

### 测试用例的约简

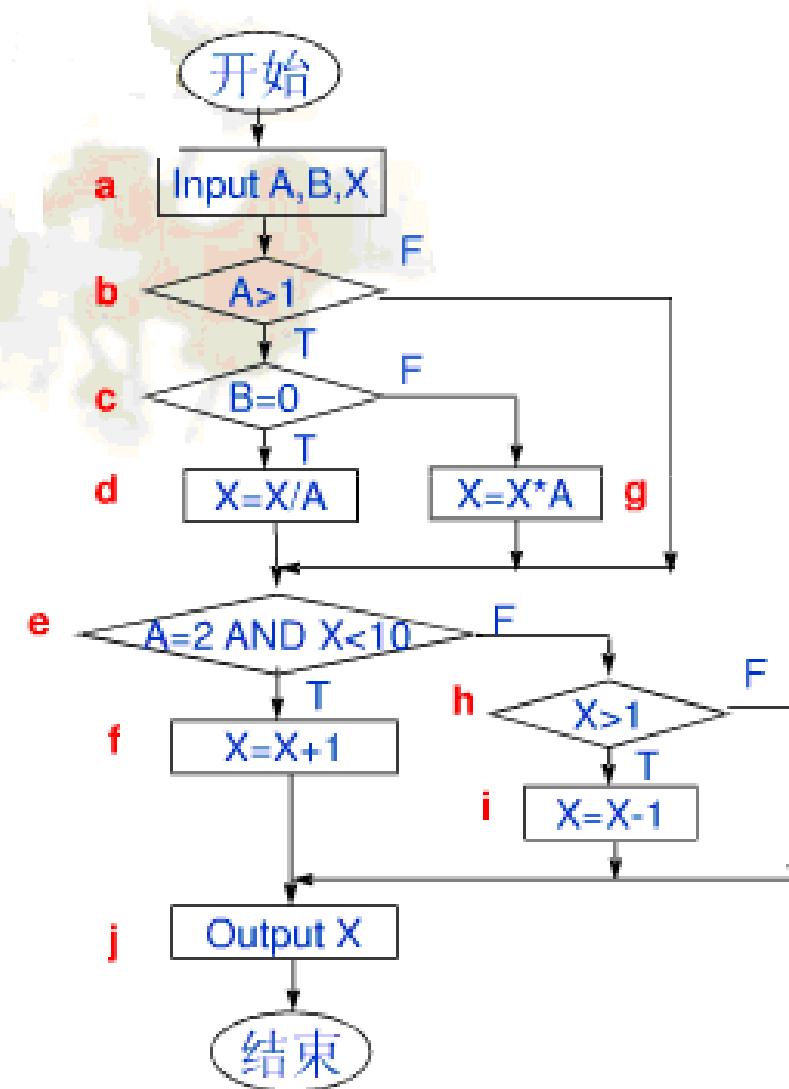
例子：利用短路效应寻找最小测试用例集

$a \&\& b \&\& (c \parallel (d \&\& e))$



#### 练习

```
START  
INPUT (A, B, X)  
IF A>1  
    THEN IF B=0  
        THEN X:=X/A  
        ELSE X:=X*A  
    ENDIF  
ENDIF  
IF (A=2 and X<10)  
    THEN X=X+1  
    ELSE IF X>1  
        THEN X=X-1  
    ENDIF  
ENDIF  
OUTPUT (X)  
STOP
```



## 动态白盒测试——路径覆盖

路径覆盖(**Path Coverage**)思想：

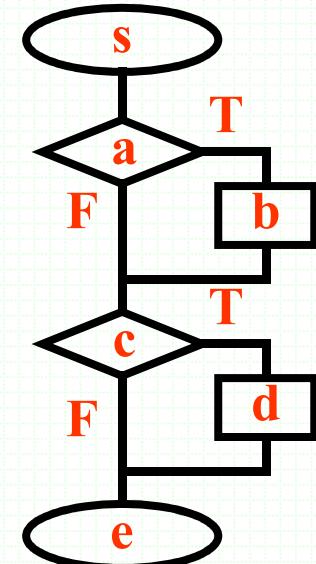
选择足够的测试用例，保证每条可能执行到的路径都至少经过一次（如果程序中包含环路，则要求每条环路至少经过一次）



廖力

- 路径测试用例  $(y, z, x)$ :

测试数据	预期结果	路径	覆盖条件	a	c
2, 0, 4	2, 0, 3	sabcde	T T T T	T	T
1, 0, 1	1, 1, 1	sace	F T F F	F	F
1, 0, 2	1, 0, 3	sacde	F T F T	F	T
3, 0, 3	3, 0, 1	sabce	T T F F	T	F

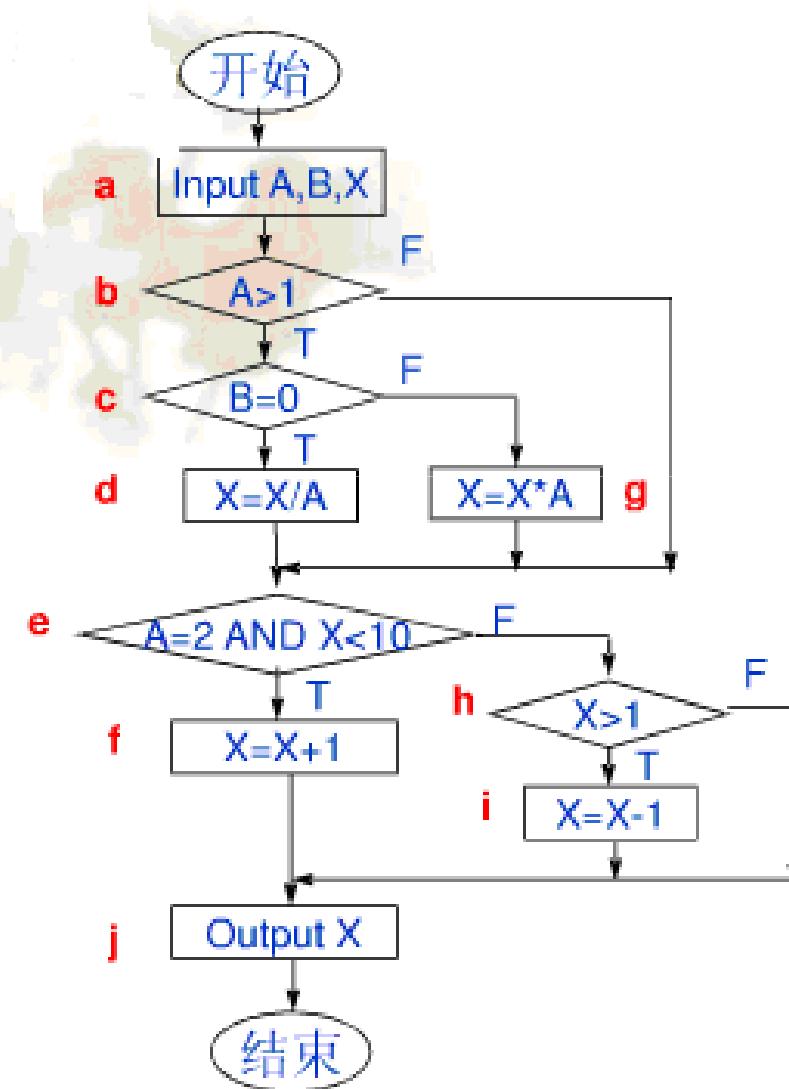


**a:**  $(y > 1) \text{ and } (z = 0)$   
**c:**  $(y = 2) \text{ or } (x > 1)$



#### 练习

```
START  
INPUT (A, B, X)  
IF A>1  
    THEN IF B=0  
        THEN X:=X/A  
        ELSE X:=X*A  
    ENDIF  
ENDIF  
IF (A=2 and X<10)  
    THEN X=X+1  
    ELSE IF X>1  
        THEN X=X-1  
    ENDIF  
ENDIF  
OUTPUT (X)  
STOP
```



## 路径覆盖(Path Coverage)优缺点：

- 优点：
  - 相对彻底的测试
- 缺点：
  - (1) 路径分支可能以指数级增加： $2^x$  ( $x$ 为分支次数)
  - (2) 不可达路径存在
  - (3) 并未测试各分支中的条件



廖力

## 动态白盒测试——路径覆盖

- 路径覆盖实际上考虑了程序中各种判定结果的所有可能组合，但它未必能覆盖判定中条件结果的各种可能情况。
- 它是一种比较强的覆盖标准，但不能替代条件覆盖和条件组合覆盖标准。



廖力

## 控制流覆盖



路径覆盖VS条件组合覆盖VS条件判定覆盖VS条件覆盖  
VS判定覆盖VS语句覆盖？

路径覆盖

条件组合覆盖

判定条件覆盖

条件覆盖

判定覆盖

语句覆盖

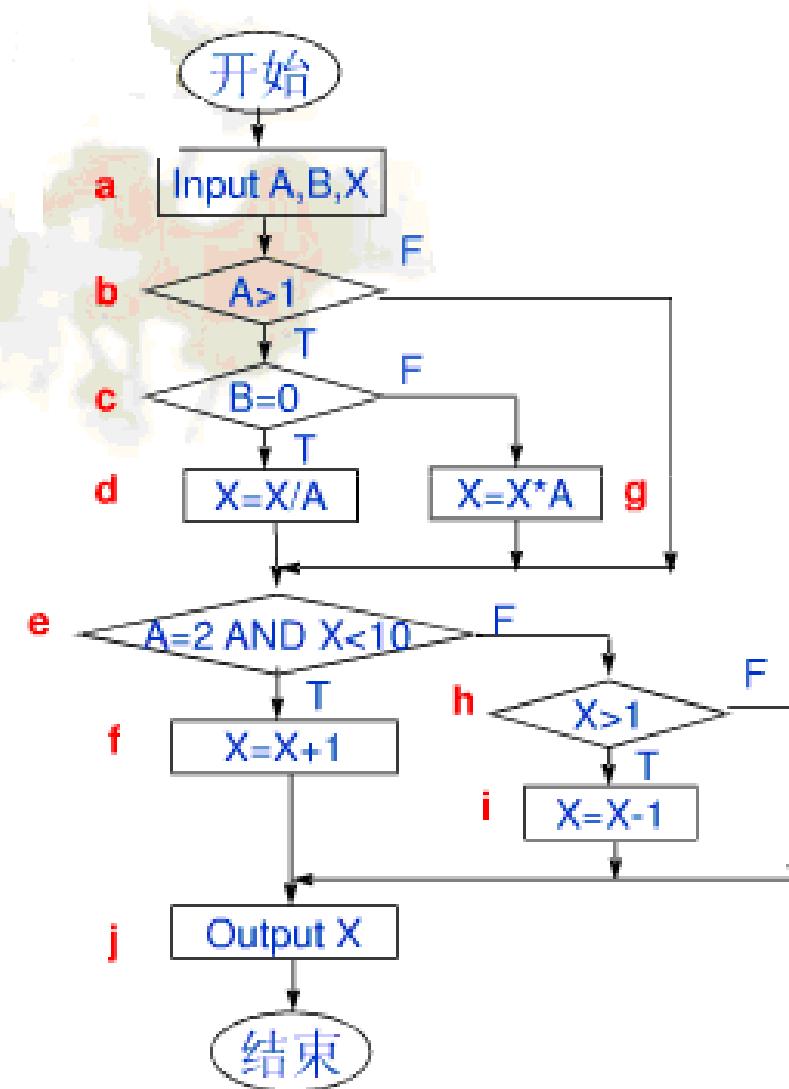
覆盖能力



廖力

## 练习

```
START  
INPUT (A, B, X)  
IF A>1  
    THEN IF B=0  
        THEN X:=X/A  
        ELSE X:=X*A  
    ENDIF  
ENDIF  
IF (A=2 and X<10)  
    THEN X=X+1  
    ELSE IF X>1  
        THEN X=X-1  
    ENDIF  
ENDIF  
OUTPUT (X)  
STOP
```



- 条件组合覆盖

测试 数据	预期结 果	覆盖的条件
A B X		B c e1 e2 h
2, 1, 3	7	T F T T /
2, 0, 24	11	T T T F T
0, 0, 1	1	F / F T F
3, 1, 4	11	T F F F T



- 路径覆盖

A	B	X	预期结果	覆盖的路径
2, 0, 8		5	abcde	fj
2, 0, 24		11	Abcde	hij
3, 0, 0		0	abcde	hj
2, 1, 3		7	abcge	fj
2, 1, 6		11	abcge	hij
3, 1, 0		0	abcgeh	j
0, 1, 9		8	abe	hij
0, 0, 1		1	abeh	j
无		无	abef	j



# 基于控制流的测试 —基本路径测试方法

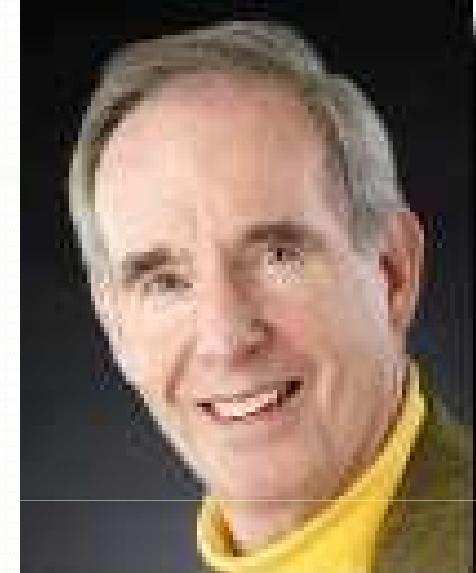


廖力

## 动态白盒测试——基本路径测试

### ■ 思想

寻找基本路径，根据基本路径构造测试用例，保证每条路径至少执行一次。



*Thomas J. McCabe*

Are you ready for  
**McCabe IQ** ?

Click here to find out.

Your Software...Better™

McCabe Software, an industry-leading Application Lifecycle Management company, provides software quality, testing, release and configuration management solutions worldwide to top commercial software, finance, defense, aerospace, healthcare, and telecommunication providers.

latest news

10/05/09 Software Pioneer and Founder, Tom McCabe, Honored with Impact Award... [more](#)

6 / 17 / 09

**McCabe IQ** has been used to analyze the quality of mission, life, and business critical software worldwide.

**McCabe CM - TRUEchange** saves you weeks with every release cycle!

New Version Our 'Integrated Difference' technology accelerates your

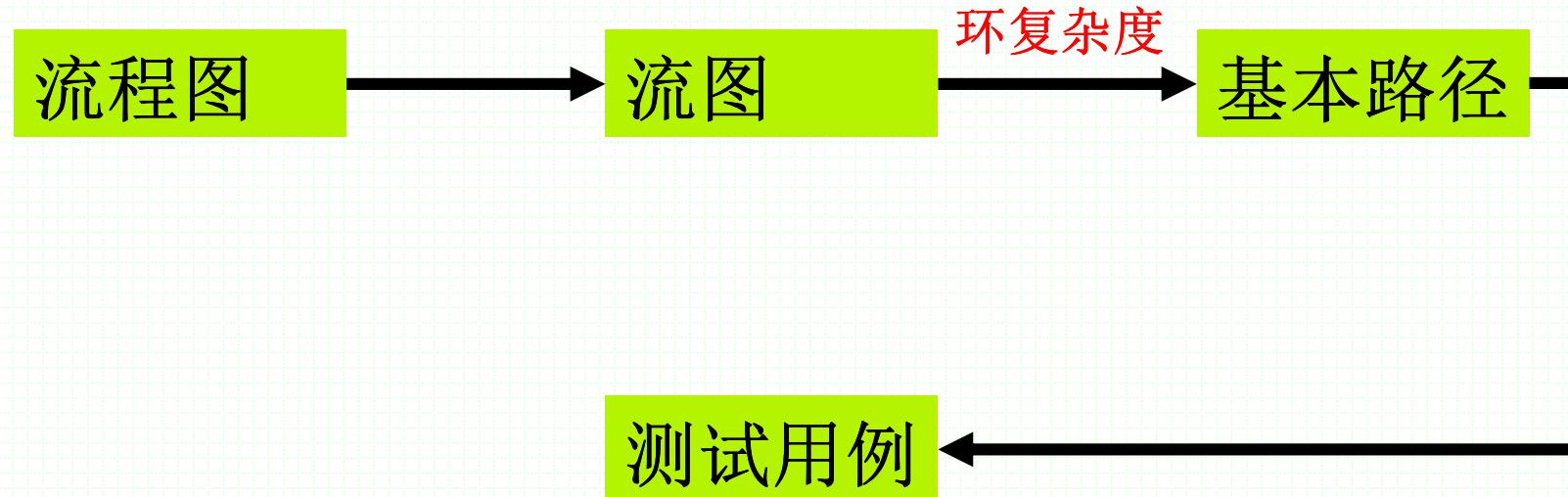


A Complexity Measure, IEEE Trans. on TSE, 1976



## 动态白盒测试——基本路径测试

### ■ 测试用例设计过程



实际上，这就是将程序流程抽象为图（连通图），通过对图的问题求解得到程序的基本路径。



廖力

## 动态白盒测试——流图

### ■ 流图

解释：

流图用来描述程序中的逻辑控制流

组成：

——结点：表示一个或多个过程语句

——边：表示控制流

——域：由边和结点限定的区间



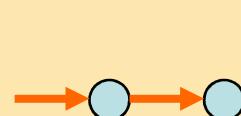
廖力

## 动态白盒测试——流图

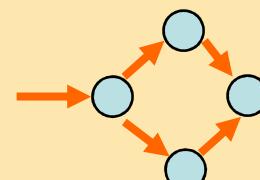
### ■ 流图

#### 流图基本结构

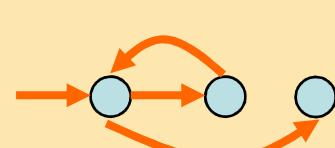
Sequence



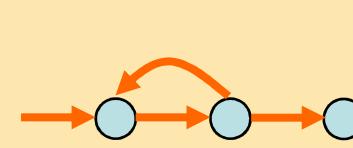
If



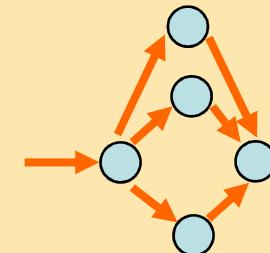
While



Until



Case

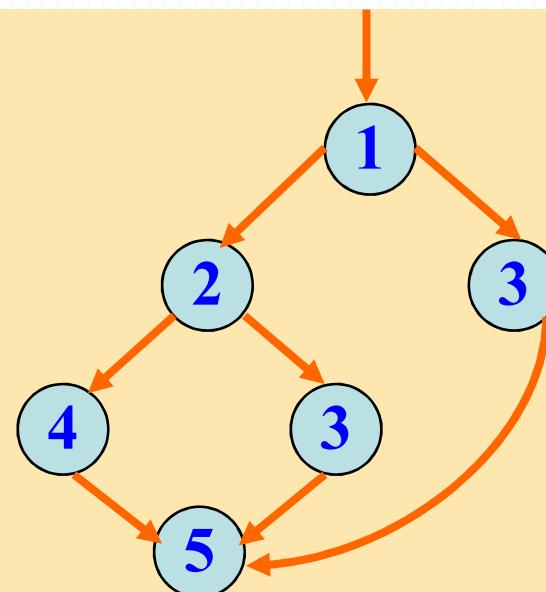


## 动态白盒测试——流图

### ■ 流图

基本结构中复合条件处理

```
2  
1 IF a OR b  
3 then procedure x  
4 Else procedure y  
5 ENDIF
```

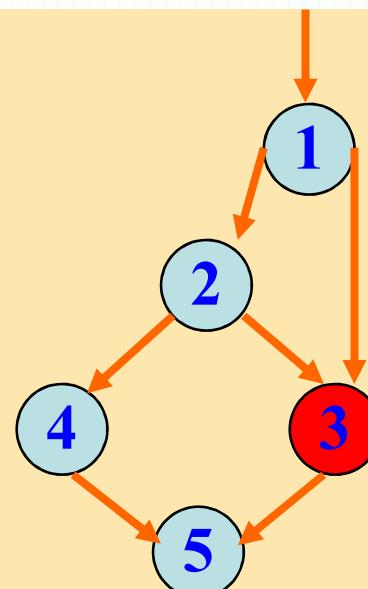


## 动态白盒测试——流图

### ■ 流图

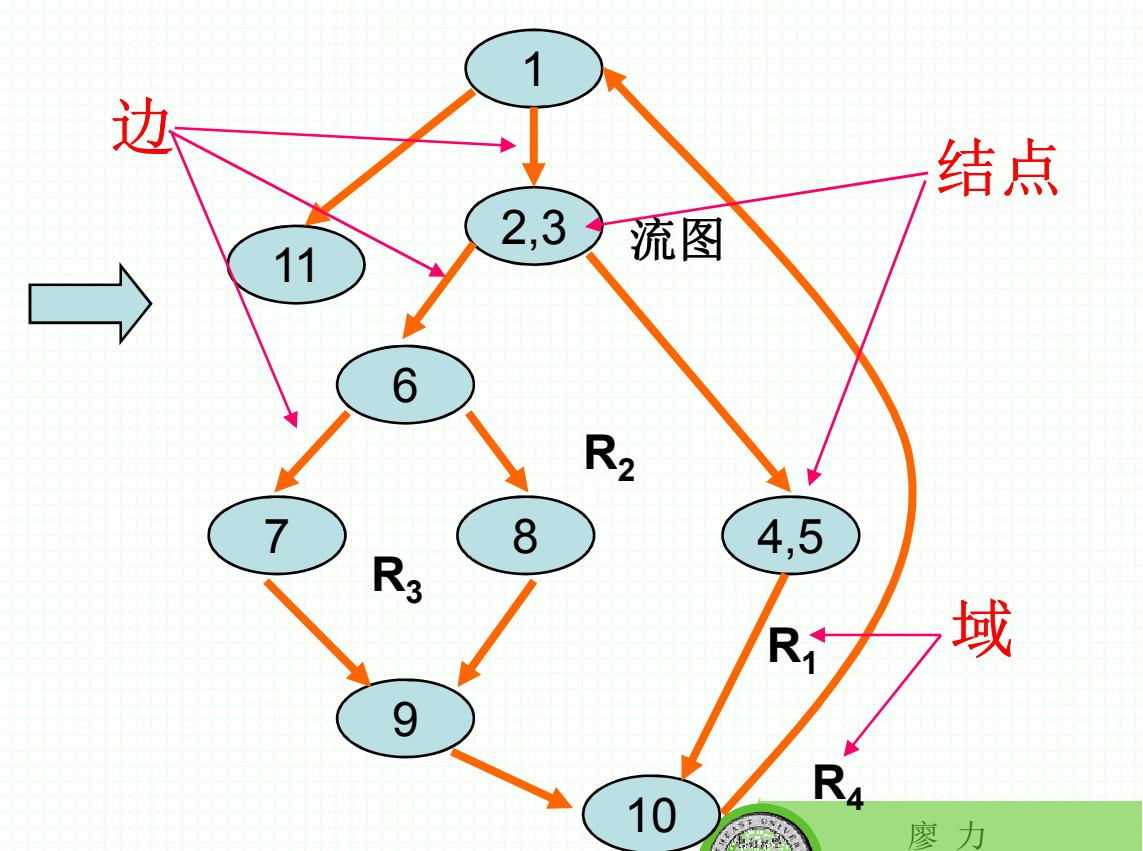
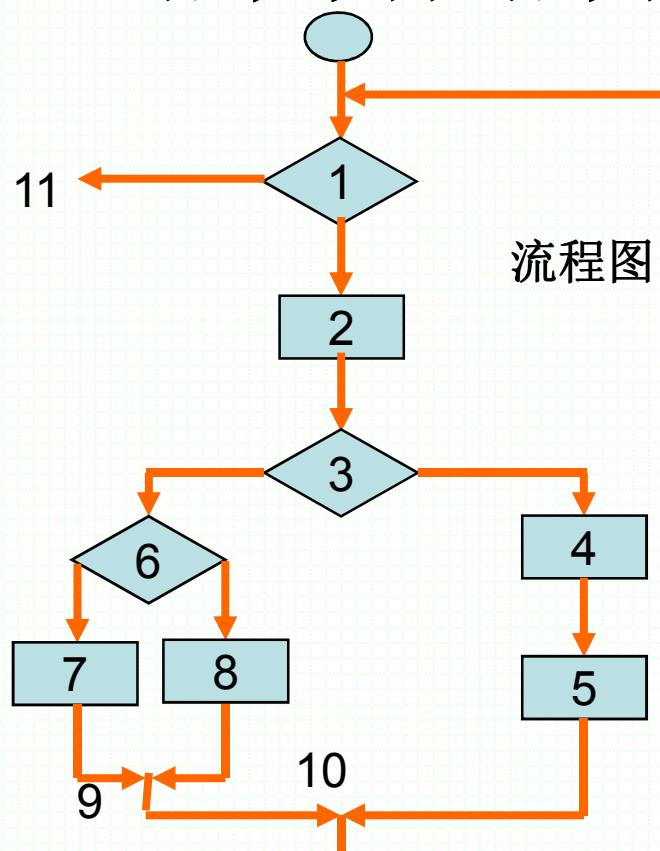
基本结构中复合条件处理

```
2  
1 IF a OR b  
3 then procedure x  
4 Else procedure y  
5 ENDIF
```



## 动态白盒测试——流图

### ■ 流程图和流图



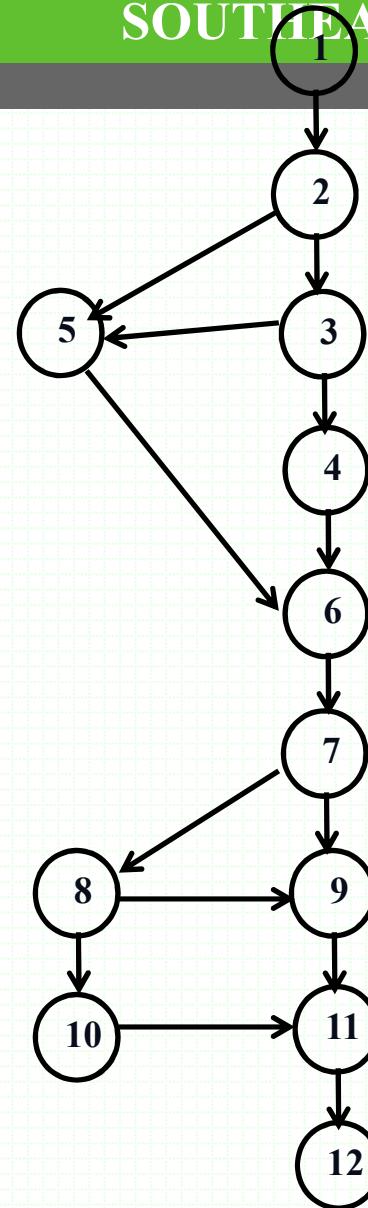
# 软件测试

Software Testing

SOUTHEAST UNIVERSITY

练习：根据PDL画出流图

```
1: start  
      input (a,b,c,d)  
2: if (a>0)  
3:   and (b>0)  
4:   then x=a+b  
5: else x=a-b  
6: endif  
7: if (c>a)  
8:   or (d<b)  
9:   then y=c-d  
10: else y=c+d  
11: endif  
12: print (x,y)  
stop
```



廖 力

## 动态白盒测试——基本路径测试

### ■ 基本路径（独立程序路径）

定义：任何贯穿程序的、至少引入一组新的处理语句或一个新判断的程序通道



廖力

## 动态白盒测试——基本路径测试

### ■ 基本路径（独立程序路径）

基本路径举例：

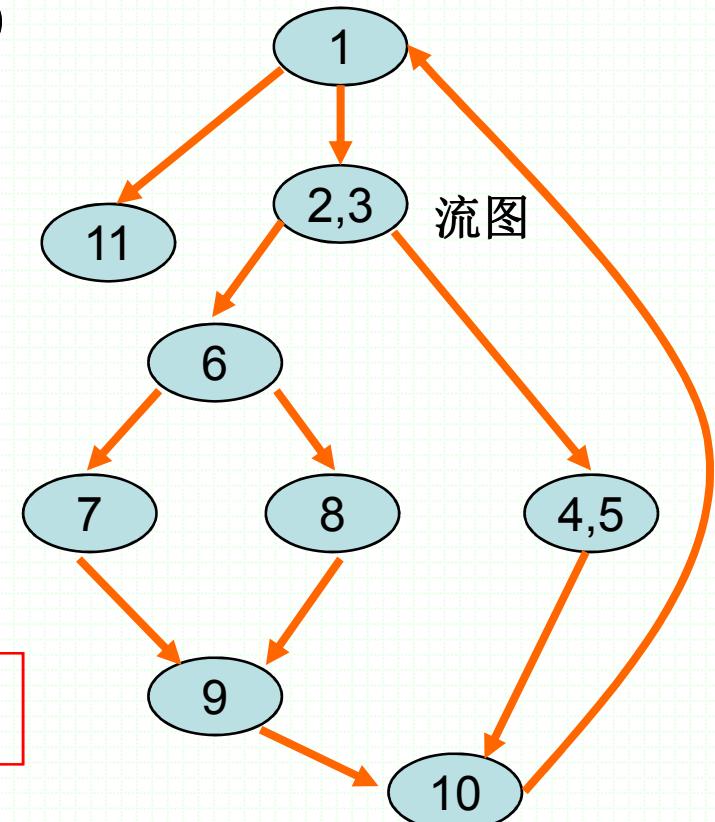
L1:1-11

L2:1-2-3-4-5-10-1-11

L3:1-2-3-6-8-9-10-1-11

L4:1-2-3-6-7-9-10-1-11

问题：可以有几条基本路径？



廖力

## 动态白盒测试——基本路径测试

### ■ 基本路径（独立程序路径）

问题：一个程序对应的基本路径？

### ■ 基本路径的度量——环复杂度

—**环复杂度**度量基本路径数，是所有语句被执行一次所需测试用例数的上限



廖力

## 动态白盒测试——基本路径测试

### ■ 基本路径（独立程序路径）

### ■ 环复杂度的计算

—方法1：等于域的数量

—方法2： $V(G) = E - N + 2$ , E为边，N为结点

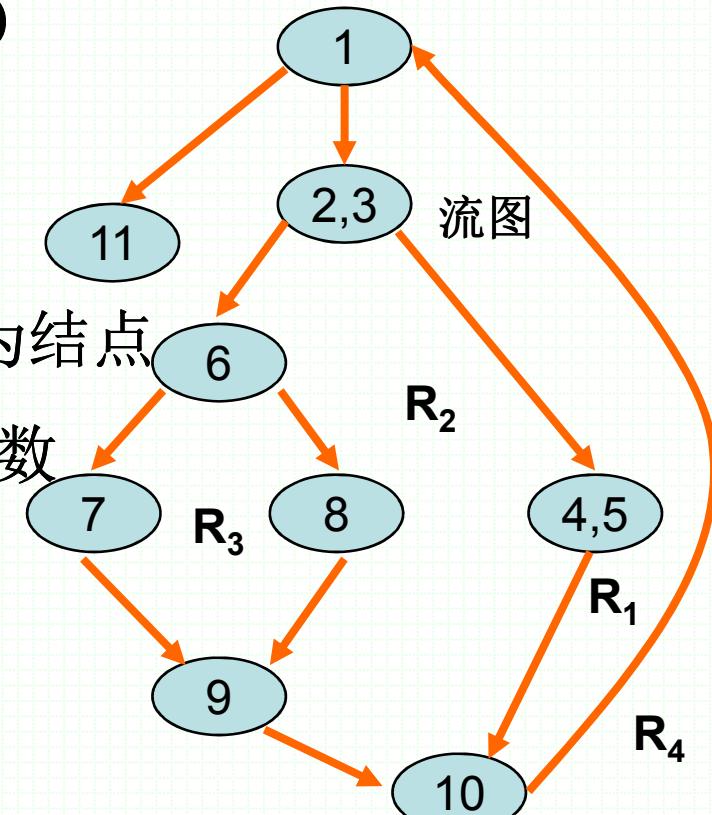
—方法3： $V(G) = P + 1$ , P为判定结点数

### ■ 实例

$$—V(G)=4$$

$$—V(G)=11-9+2=4$$

$$—V(G)=3+1=4$$

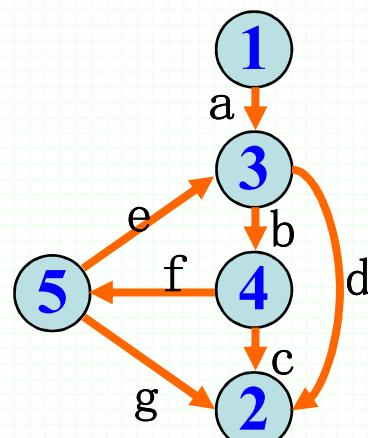


## 动态白盒测试——基本路径测试

### ■ 基本路径（独立程序路径）——环复杂度的计算

图矩阵法

作用：测试自动化中的一种有用的数据结构



	1	2	3	4	5
1			a		
2					
3		d		b	
4	c				f
5	g	e			

与流图对应

行列——节点数

项——边

判定节点——多个节点的行数

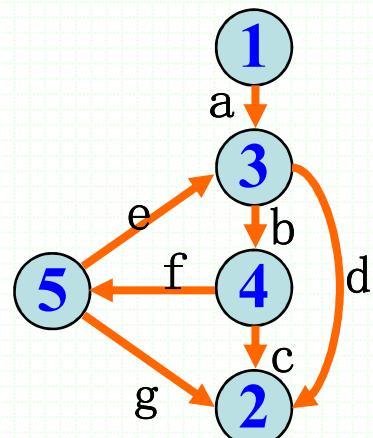


廖力

## 动态白盒测试——基本路径测试

### ■ 基本路径（独立程序路径）——环复杂度的计算

图矩阵法（连接矩阵）



	1	2	3	4	5
1			1		
2					
3		1		1	
4	1				1
5	1	1			

连接  
 $1-1=0$   
 $2-1=1$   
 $2-1=1$   
 $2-1=1$

环复杂度= $3+1=4$



环复杂度	对应含义
1-10	代码编写良好，有很高的可测试性，维护成本低
11-20	中等复杂度，可测试性中等，维护成本中等
21-50	非常复杂，可测试性低，维护成本高
>50	没人看得懂

具有最高环复杂度的模块蕴含错误的可能性最大，是测试中关注的焦点



廖力

## 例如

Jeditor0.2项目的方法总数为550个，其中：

方法圈复杂度(cc) 范围	方法个数	所占比例
cc<10	510	92.72%
10<=cc<20	22	4%
20<=cc<50	9	1.64%
cc>=50	9	1.64%



廖力

#### 超过阈值的类列表

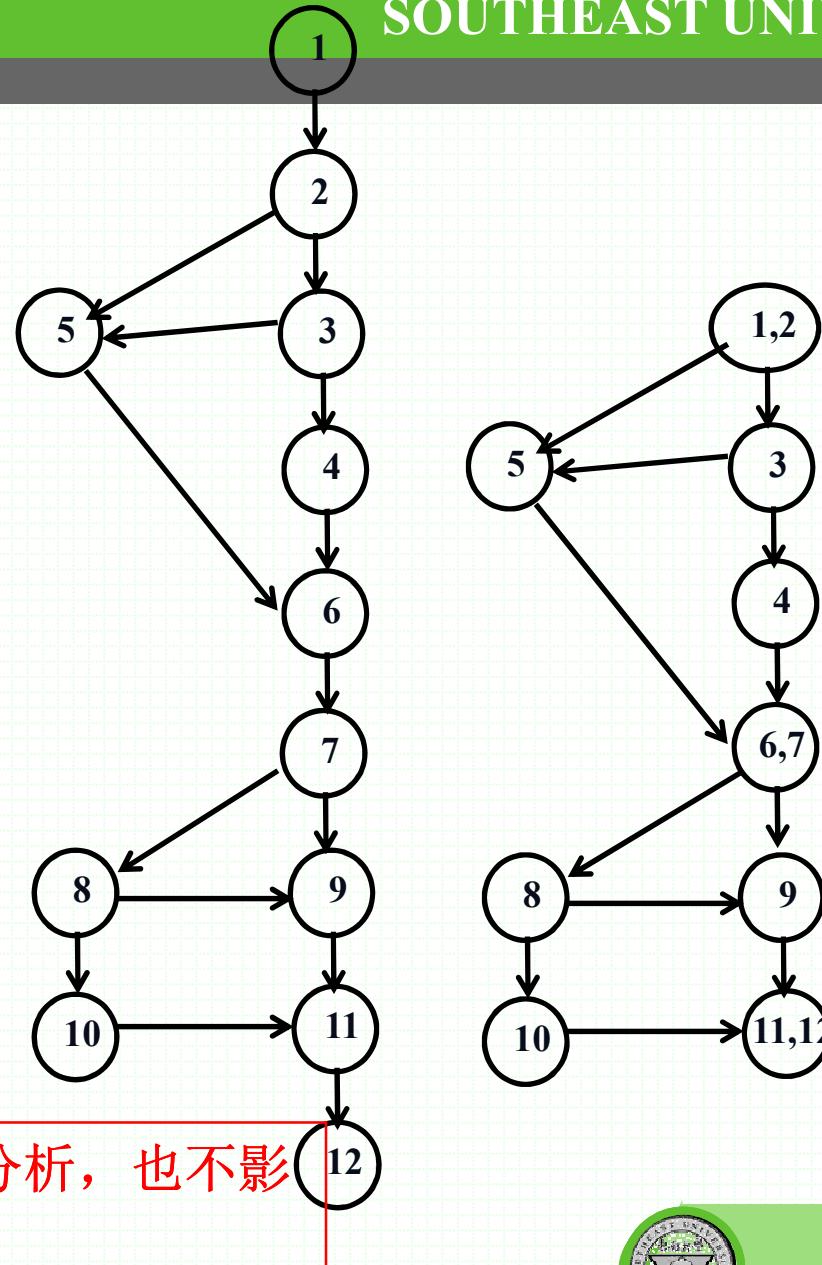
包名	类名	圈复杂度
org.jeditor.gui	JEditor	353
org.jeditor.scripts	PerlTokenMarker	161
org.jeditor.gui	CodeEditorPainter	104
org.jeditor.diff	Diff	93
org.jeditor.scripts	FortranTokenMarker	86



廖力

根据PDL画出流图

```
1: start  
    input (a,b,c,d)  
2: if (a>0)  
3:     and (b>0)  
4:     then x=a+b  
5:     else x=a-b  
6: endif  
7: if (c>a)  
8:     or (d<b)  
9:     then y=c-d  
10:    else y=c+d  
11: endif  
12: print (x,y)
```



顺序节点的合并并不影响环复杂度分析，也不影响基本路径获取。  
stop

廖力



## 动态白盒测试——基本路径测试

■ 基本路径（独立程序路径）

■ 基本路径集合

$P_1: 1 \rightarrow 11$

$P_2: 1 \rightarrow 2, 3 \rightarrow 6 \rightarrow 7 \rightarrow 9 \rightarrow 10 \rightarrow 1 \rightarrow 11$

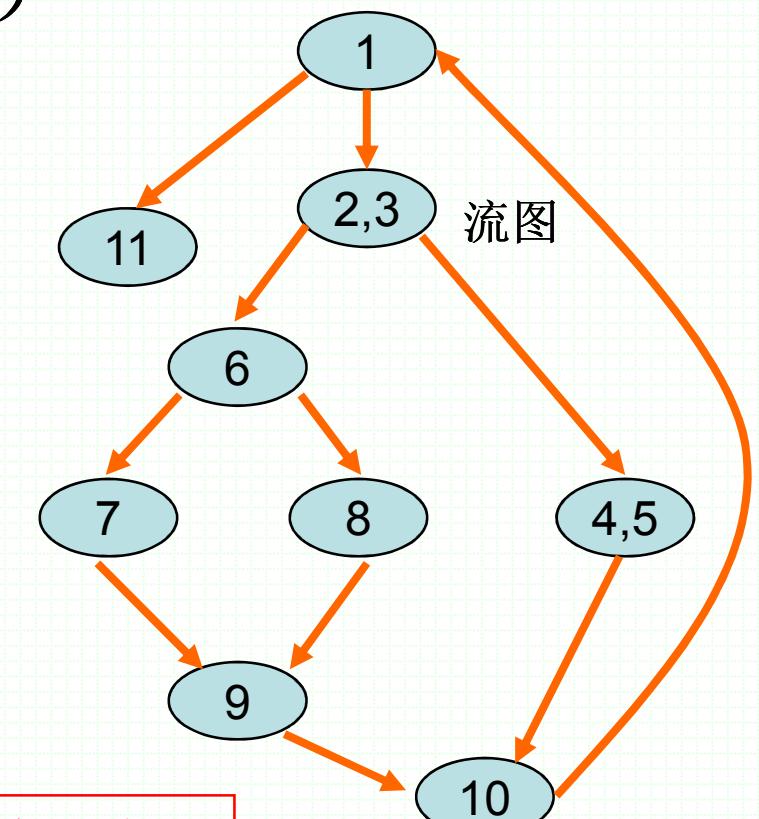
$P_3: 1 \rightarrow 2, 3 \rightarrow 4, 5 \rightarrow 10 \rightarrow 1 \rightarrow 11$

$P_4: 1 \rightarrow 2, 3 \rightarrow 6 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 1 \rightarrow 11$

新路径特点：贯穿程序；

引入新的处理语句或新判断

基本路径集合并不唯一，但路径数是确定的



## 基本路径测试分析

### ■ 基本路径集寻找算法？

**Step1:** 确认从入口到出口的最短基本路径

**Step2:** 从入口到第1个未被先后评估为真和假两种结果的条件语句

**Step3:** 改变该条件语句的判断值

**Step4:** 按最短路径从这个条件语句到出口

**Step5:** 重复步骤2-5,直到所有基本路径都被找到

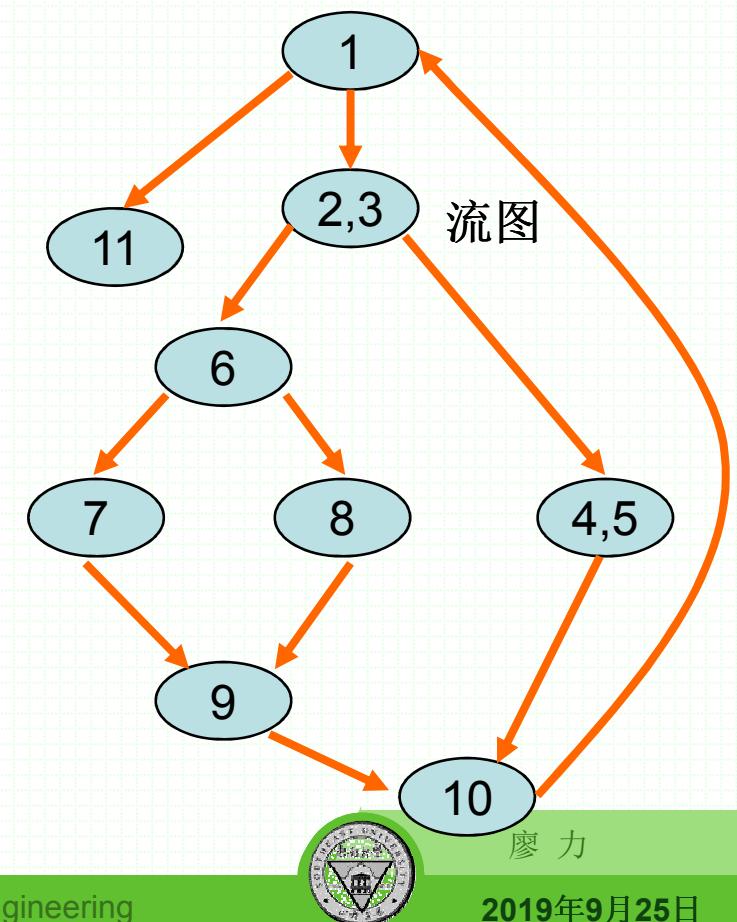


廖力

## 动态白盒测试——基本路径测试

### ■ 基本路径（独立程序路径）

```
void Func(int nPosX, int nPosY) {  
    1   while (nPosX > 0) {  
    2       int nSum = nPosX + nPosY;  
    3       if (nSum > 1) {  
    4           nPosX--;  
    5           nPosY--;  
    }  
    else {  
    6,7        if (nSum < -1) nPosX -= 2;  
    8        else nPosX -= 4;  
    9    }  
    10   }// end of while  
    11}
```



## 动态白盒测试——基本路径测试

### ■ 基本路径（独立程序路径）

$P_1: 1 \rightarrow 11$

✓ 用例1: nPosX 取-1, nPosY取任意值

$P_2: 1 \rightarrow 2, 3 \rightarrow 6 \rightarrow 7 \rightarrow 9 \rightarrow 10 \rightarrow 1 \rightarrow 11$

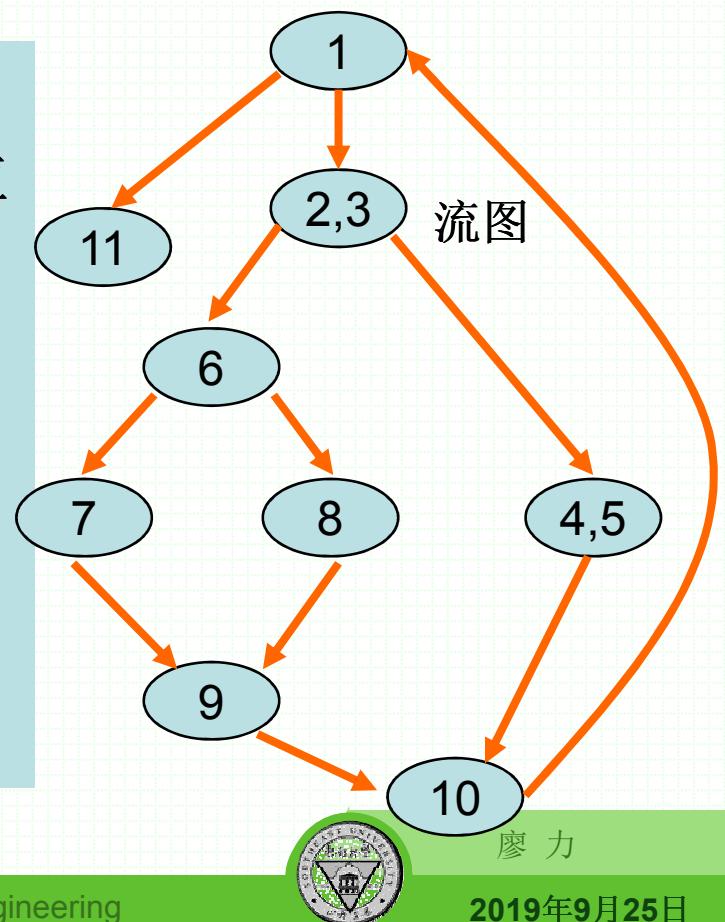
✓ 用例2: nPosX 取1, nPosY取1

$P_3: 1 \rightarrow 2, 3 \rightarrow 4, 5 \rightarrow 10 \rightarrow 1 \rightarrow 11$

✓ 用例3: nPosX 取1, nPosY取-1

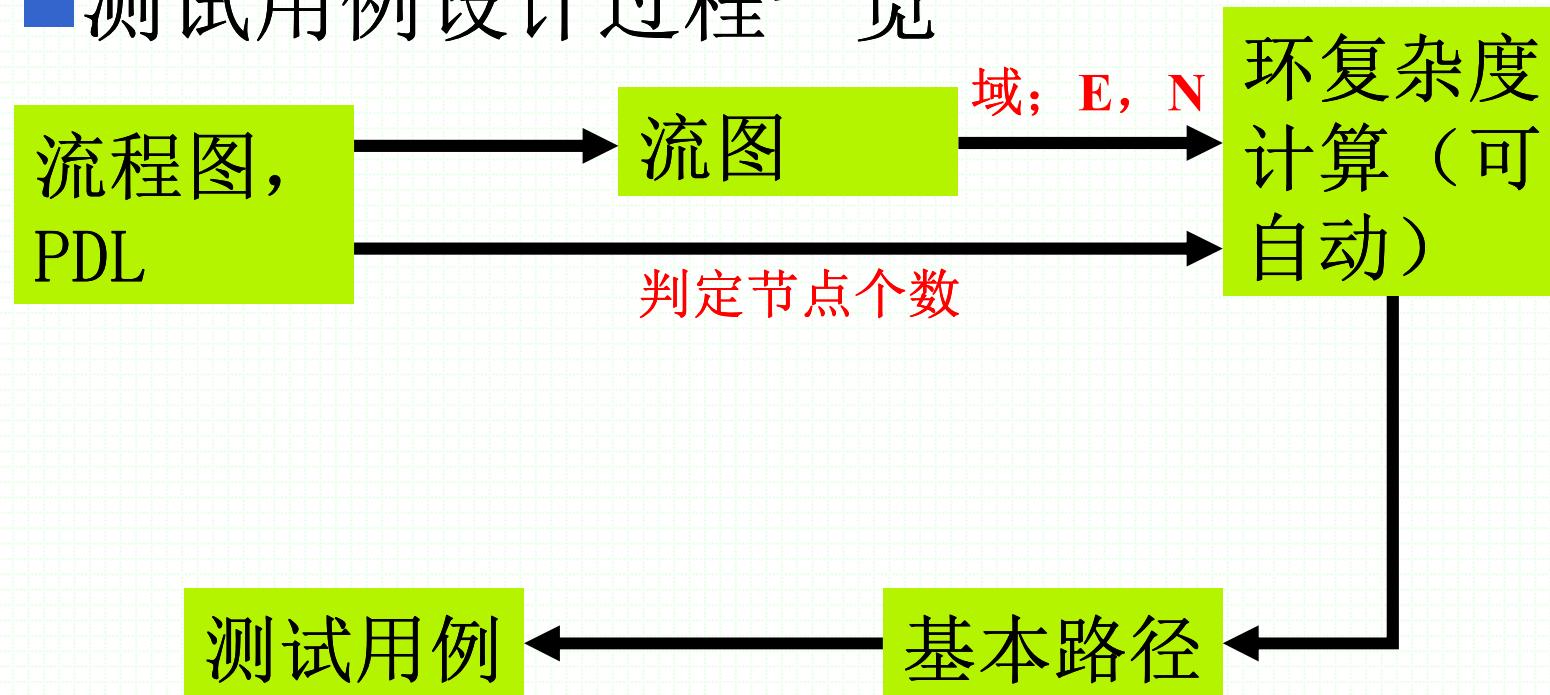
$P_4: 1 \rightarrow 2, 3 \rightarrow 6 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 1 \rightarrow 11$

✓ 用例4: nPosX 取1, nPosY取-3



## 动态白盒测试——基本路径测试

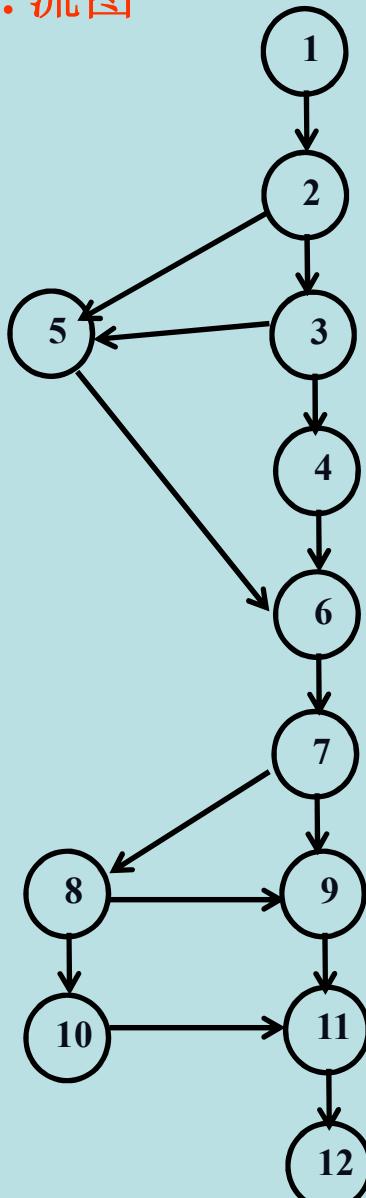
### ■ 测试用例设计过程一览



# 基本路径测试练习1

```
1: start  
    input (a,b,c,d)  
2: if (a>0)  
3:     and (b>0)  
4: then x=a+b  
5: else x=a-b  
6: endif  
7: if (c>a)  
8:     or (d<b)  
9: then y=c-d  
10: else y=c+d  
11: endif  
12: print (x,y)  
stop
```

## 1. 流图



## 2. 圈复杂度

$$V(G)=5$$

## 3. 基本路径

p1: 1,2,5,6,7,9,11,12

p2: 1,2,5,6,7,8,9,11,12

p3: 1,2,5,6,7,8,10,11,12

p4: 1,2,3,5,6,7,9,11,12

p5: 1,2,3,4,6,7,9,11,12

## 4. 测试用例

p1: 1,2,5,6,7,9,11,12

a=-1, b=1, c=1, d=1

p2: 1,2,5,6,7,8,9,11,12

a=-1, b=1, c=-1, d=0

p3: 1,2,5,6,7,8,10,11,12

a=-1, b=1, c=-1, d=1

p4: 1,2,3,5,6,7,9,11,12

a=1, b=-1, c=2, d=1

p5: 1,2,3,4,6,7,9,11,12

a=1, b=1, c=2, d=1

## 动态白盒测试——基本路径测试

### ■ 基本路径（独立程序路径）

问题1：为什么流图中会有一个开放的域？

问题2：能否精确定义基本路径？

问题3：基本路径集合是怎么得到的？基本路径的获取有没有顺序性？



廖力

## 动态白盒测试——基本路径测试

■为什么流图中会有一个开放的域？

基本路径测试理论是建立在程序图是一个强连通图的基础上。

**强连通图：**在有向图G中，若对于每一对图中节点 $v_i$ 和 $v_j$ ，从 $v_i$ 到 $v_j$ 和从 $v_j$ 到 $v_i$ 都存在路径，则G为强连通图。



廖力

## 动态白盒测试——基本路径测试

### ■为什么流图中会有一个开放的域？

显然，程序图不可能是一个强连通图。

因此，基本路径测试理论中，引入了一条虚拟边，从程序出口点指向程序入口点，从而构成强连通图。

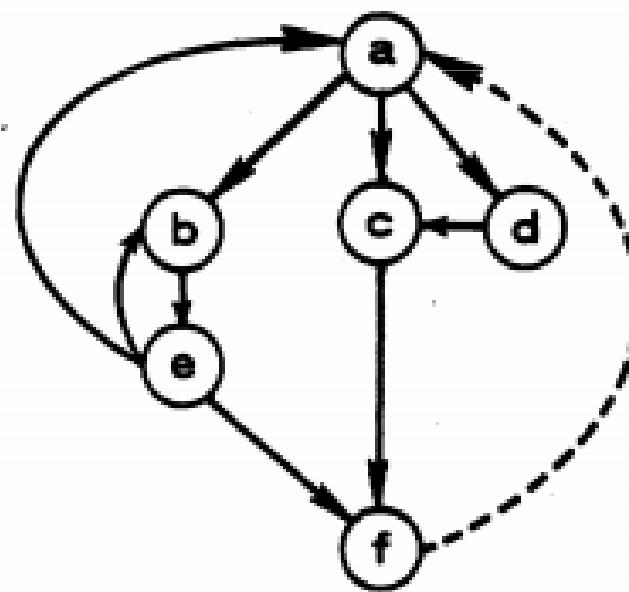
正是这条虚拟边构成了最后一个域，也就是教材上说的开放域。



廖力

## 动态白盒测试——基本路径测试

■ 为什么流图中会有一个开放的域？



廖力

## 基本路径测试分析

### ■ 基本路径的本质？

每一条路径都可以表示为边的向量，即  
边作为向量的维；

$$P_x = (e_1, e_2, \dots, e_n), \quad n = EdgeSize$$

如果路径经过边1次，向量值对应的维就是1，如果经过2次，向量值就是2，以此类推。



廖力

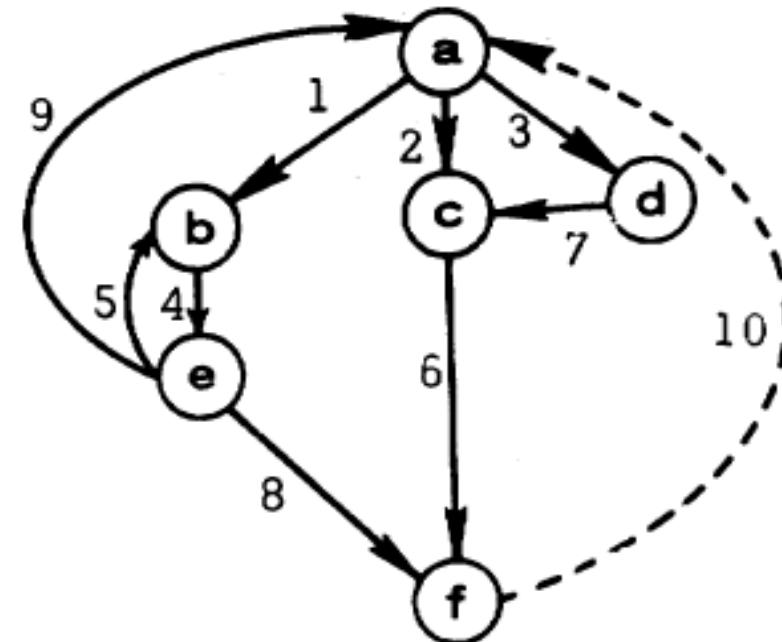
## 基本路径测试分析

### ■ 基本路径的本质？

任意给定一条路径:  $P_0 = a \rightarrow b \rightarrow e \rightarrow b \rightarrow e \rightarrow b \rightarrow e \rightarrow f$

则它对应的向量为:

Edge: 1	2	3	4	5	6	7	8	9
P <sub>0</sub> :	1	0	0	3	2	0	0	1



## 基本路径测试分析

### ■ 基本路径的数学定义

所谓基本路径，是指一组路径集合，其它任何路径对应的向量，都可以用这些基本路径对应的向量通过线性运算组合出来。



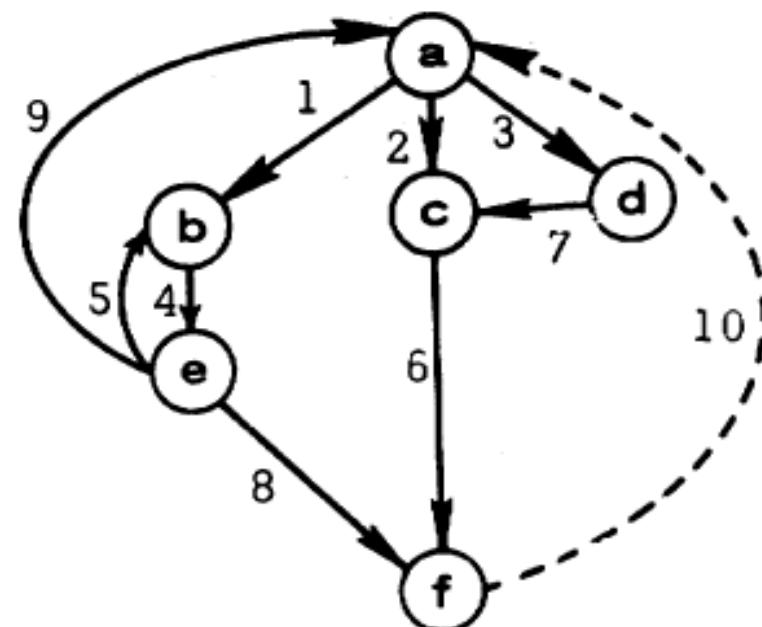
廖力

## 基本路径测试分析

### ■ 基本路径的本质？

我们就有如下一个基本路径集合：

- $P_1:$  a c f
- $P_2:$  a d c f
- $P_3:$  a b e f
- $P_4:$  a b e b e f
- $P_5:$  a b e a c f



## 基本路径测试分析

### ■ 基本路径的本质？

对应的向量分别是：

Edge:	1	2	3	4	5	6	7	8	9
P <sub>1</sub> :	0	1	0	0	0	1	0	0	0
P <sub>2</sub> :	0	0	1	0	0	1	1	0	0
P <sub>3</sub> :	1	0	0	1	0	0	0	1	0
P <sub>4</sub> :	1	0	0	2	1	0	0	1	0
P <sub>5</sub> :	1	1	0	1	0	1	0	0	1



## 基本路径测试分析

### ■ 基本路径的本质？

任何其它路径都可以通过向量组合运算出来：

例如路径

$P_0: 1 \ 0 \ 0 \ 3 \ 2 \ 0 \ 0 \ 1 \ 0$

$$P_0 = x_1 * P_1 + x_2 * P_2 + x_3 * P_3 + x_4 * P_4 + x_5 * P_5$$

经过求解，我们可以得到：  $P_0 = 2 * P_4 - P_3$



廖 力

## 基本路径测试分析

### ■ 基本路径集合不唯一

基本路径集合的大小是由环复杂度确定的，但基本路径的组成并不确定。

$P_1: \text{a c f}$

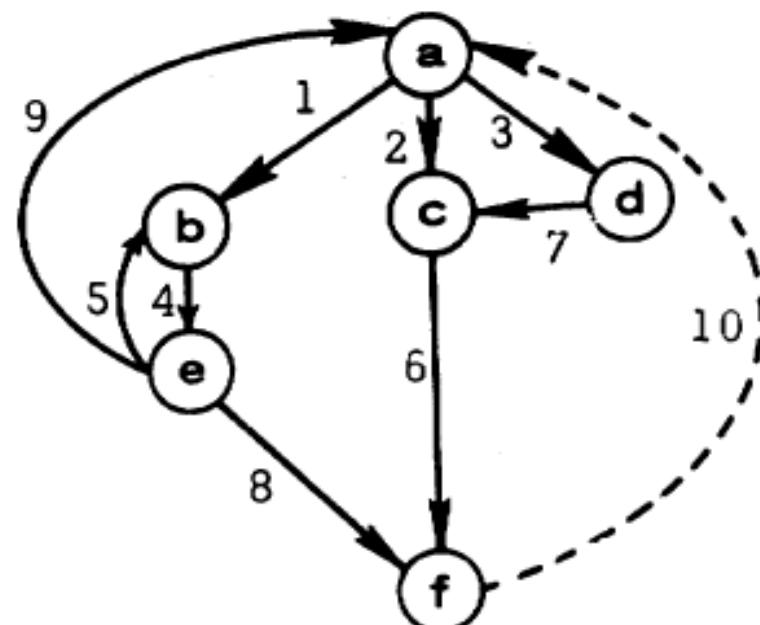
$P_2: \text{a d c f}$

$P_3: \text{a b e f}$

$P_4: \text{a b e b e f}$

$P_5: \text{a b e a c f}$

$P_{5\text{new}}: \text{a b e a d c f}$



## 基本路径测试分析

■ 基本路径集合不唯一  
对应的向量是：

Edge:	1	2	3	4	5	6	7	8	9
P <sub>1</sub> :	0	1	0	0	0	1	0	0	0
P <sub>2</sub> :	0	0	1	0	0	1	1	0	0
P <sub>3</sub> :	1	0	0	1	0	0	0	1	0
P <sub>4</sub> :	1	0	0	2	1	0	0	1	0
P <sub>5</sub> :	1	1	0	1	0	1	0	0	1
P <sub>5new</sub> :	1	0	1	1	0	1	1	0	1



## 基本路径测试分析

### ■ 基本路径集合不唯一

$P_0:$  1 0 0 3 2 0 0 1 0

$$\begin{aligned}P_0 &= x_1 * P_1 + x_2 * P_2 + x_3 * P_3 + x_4 * P_4 + x_5 * P_5_{\text{new}} \\&= 2 * P_4 - P_3\end{aligned}$$

$P:$  abebeadcf

$P:$  1 0 1 2 1 1 1 0 1

$$P = P_5_{\text{new}} + P_4 - P_3$$

$$P = P_5 + P_2 + P_4 - P_3 - P_1$$



廖力

## 基本路径测试分析

### ■ 基本路径集合不唯一

新的基本路径集合和前面的集合相比，只是最后一个路径不同。但是这5条路径中都是线性无关的，而且，其它的路径都可以通过他们组合生成，因此也是基本路径集合。



廖力

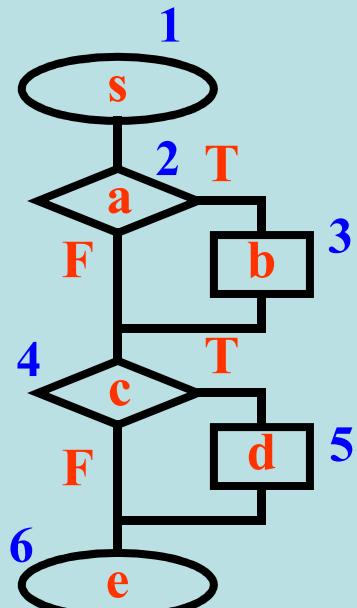
练习：对下列子程序进行基本路径测试

```
procedure example(y,z:real;var x:real);
begin
  if (y>1) and (z=0) then x:=x/y;
  if (y=2) or (x>1) then x:=x+1;
end;
```



廖力

# 基本路径测试练习1



b:  $x = x/y$

d:  $x = x + 1$

2

2x

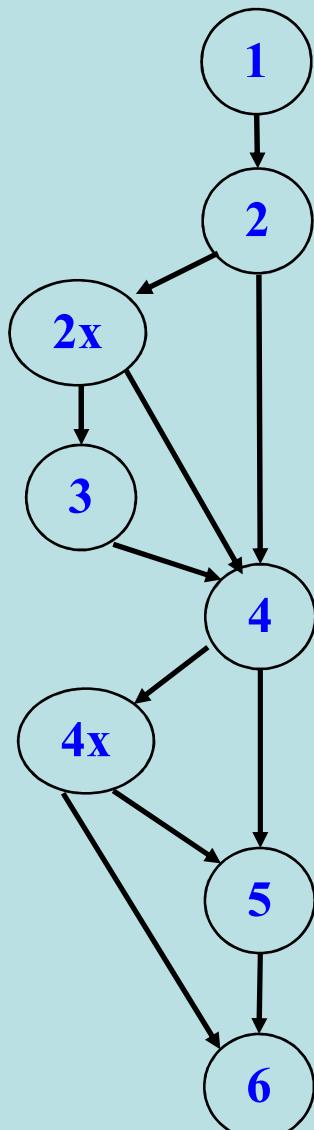
a:  $(y > 1) \text{ and } (z = 0)$

c:  $(y = 2) \text{ or } (x > 1)$

4

4x

## 1. 流图



## 2. 圈复杂度

$$V(G) = 5$$

## 3. 基本路径

p1: 1,2,4,5,6

p2: 1,2,4,4x,6

p3: 1,2,4,4x,5,6

p4: 1,2,2x,4,5,6

p5: 1,2,2x,3,4,5,6

## 4. 测试用例

p1: 1,2,4,5,6

x=0, y=?, z=1 (无测试用例)

不可达路径

p2: 1,2,4,4x,6

x=0, y=1, z=1

p3: 1,2,4,4x,5,6

x=2, y=1, z=1

p4: 1,2,2x,4,5,6

x=0, y=2, z=1

p5: 1,2,2x,3,4,5,6

x=0, y=2, z=0

## 基本路径测试分析

### ■ 基本路径集寻找算法？

**Step1:** 确认从入口到出口的最短基本路径

**Step2:** 从入口到第1个未被先后评估为真和假两种结果的条件语句

**Step3:** 改变该条件语句的判断值

**Step4:** 按最短路径从这个条件语句到出口

**Step5:** 重复步骤2-5,直到所有基本路径都被找到



廖力

# 基于控制流的测试 —循环的处理



廖力

## 动态白盒测试

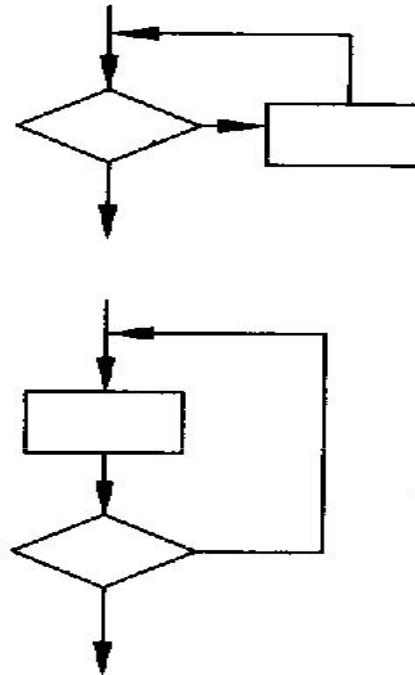
### ■ 循环的处理

循环分为4种不同类型：

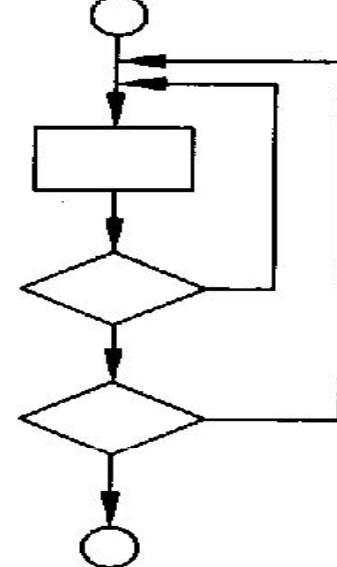
- 简单循环
- 嵌套循环
- 串接循环
- 非结构循环



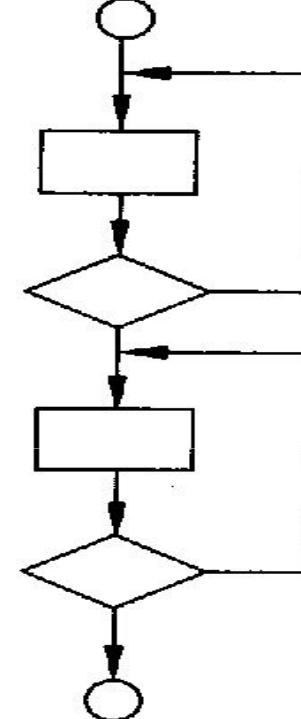
廖力



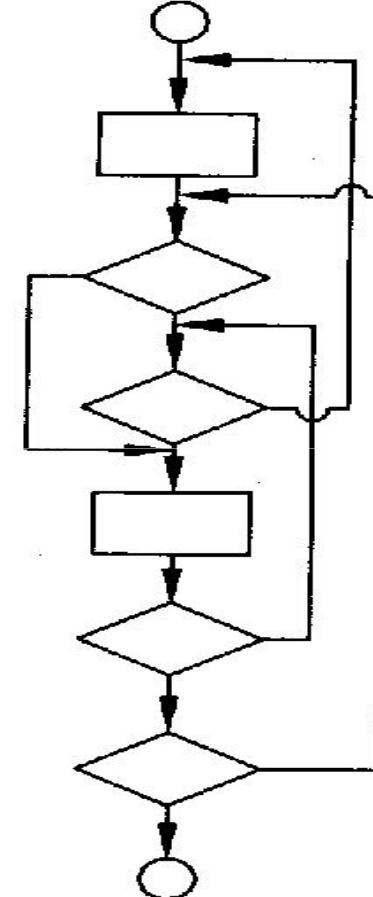
(a) 简单循环



(b) 嵌套循环



(c) 串接循环



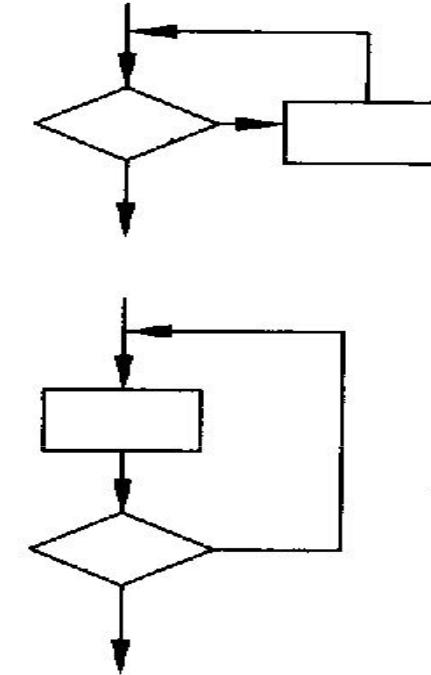
(d) 非结构循环



## 控制流测试—循环的处理

### ■ 简单循环的测试用例构造

- (1) 跳过整个循环
- (2) 只执行一次循环
- (3) 执行两次循环
- (4) 执行 $m (m < n)$ 次循环
- (5) 执行 $n-1, n, n+1$ 次循环



(a) 简单循环

## 控制流测试—循环的处理

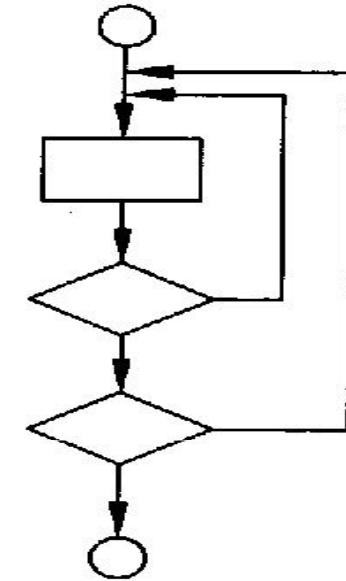
### ■ 嵌套循环的测试用例构造

① 先测试最内层循环：

- 所有外层的循环变量置为最小值
- 最内层按简单循环测试；

② 由里向外，测试上层循环：

- 此层以外的所有外层循环的循环变量取最小值
- 此层以内的所有嵌套内层循环的循环变量取“典型值”
- 该层按简单循环测试；



(b) 嵌套循环

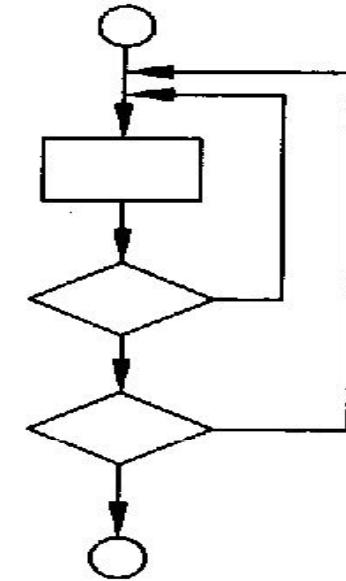


廖力

## 控制流测试—循环的处理

### ■ 嵌套循环的测试用例构造

- ① 先测试最内层循环
- ② 由里向外，测试上一层循环
- ③ 重复上一条规则，直到所有各层循环测试完毕；
- ④ 对全部各层循环同时取最小循环次数，或者同时取最大循环次数



(b) 嵌套循环



廖力

## 控制流测试—循环的处理

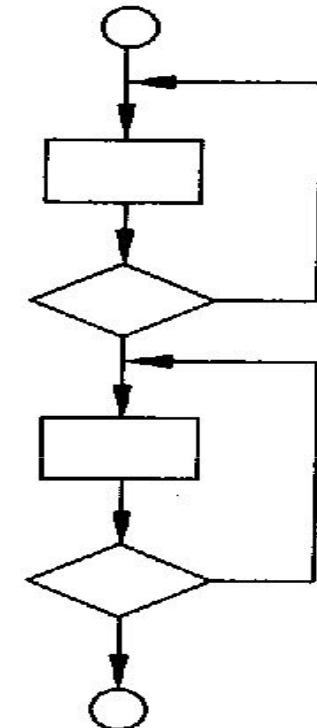
### ■ 串接循环的测试用例构造

① 若串接的各个循环互相独立：

- 分别用简单循环的方法进行测试；

② 若两个循环不独立（第一个循环的循环变量与第二个循环控制相关）

- 把第一个循环看作外循环，第二个循环看作内循环，然后用测试嵌套循环的办法来处理。



(c) 串接循环

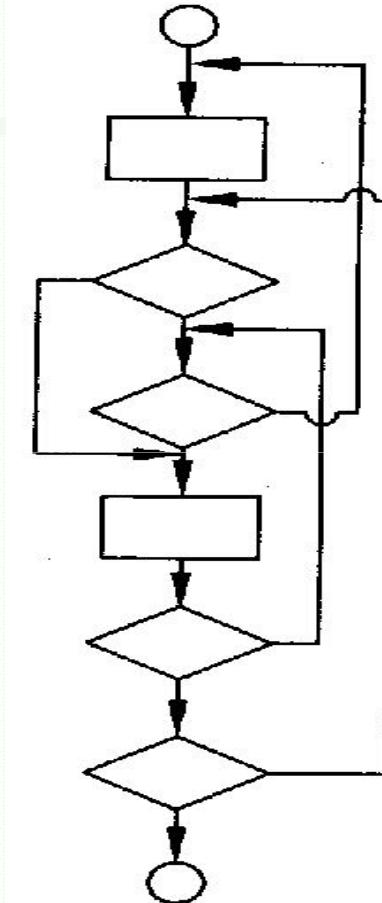


廖力

## 控制流测试—循环的处理

### ■ 非结构循环

这一类循环应该先将其结构化  
，然后再测试。



(d) 非结构循环

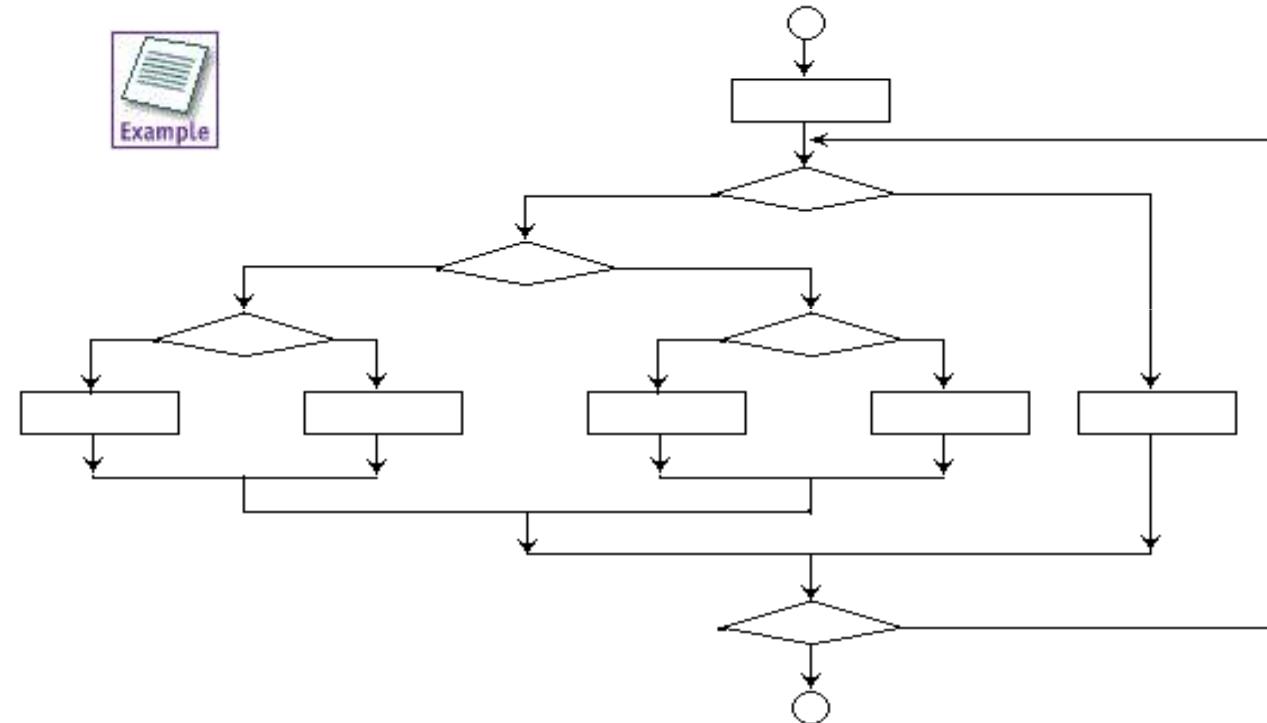


廖力

## ■ 穷举测试？



Example



循环  $\leq 20$  次



廖 力

# 基于数据流的测试



廖力

东南大学 软件学院 College of Software Engineering

2019年9月25日

## 动态白盒测试——数据流覆盖

### ■ 数据流覆盖思想

1. 程序是对**数据的加工处理**过程，对程序的测试可从数据处理流程的角度进行考虑。

2. Debugging的启发：

寻找关于某变量的**定义和使用位置**，思考程序在运行时该变量的值会如何变化，从而分析Bug产生的原因。

3. 数据的处理过程**对应一定的控制流路径**



廖 力

```
1 Program Example()
2 var staffDiscount, totalPrice, finalPrice, discount, price
3 staffDiscount = 0.1
4 totalPrice = 0
5 input(price)
6 while(price != -1) do
7   totalPrice = totalPrice + price
8   input(price)
9 od
10 print("Total price: " + totalPrice)
11 if(totalPrice>15.00) then
12   discount = (staffDiscount * totalPrice) + 0.50
13 else
14   discount = staffDiscount * totalPrice
15 fi
16 print ("Discount: " + discount)
17 finalPrice = totalPrice - discount
18 print ("Final price: " + finalPrice)
19 endProgram
```

## 动态白盒测试——数据流覆盖

### ■ 数据流测试原理

根据程序中变量定义和其后变量使用的位置来选择程序的测试路径

### ■ 数据流覆盖常用标准

□ 1. Rapps 和 Weyuker 标准

□ 2. Ntafos 标准

□ 3. Ural 标准

□ 4. Laski 和 Korel 标准



廖力

## 动态白盒测试——数据流覆盖

### ■ 数据流覆盖基础

基本定义：

1. **P**——程序
2. **G(P)**——程序图（流图）
3. **V**——变量集合
4. **PATH(P)**——P的所有路径集合

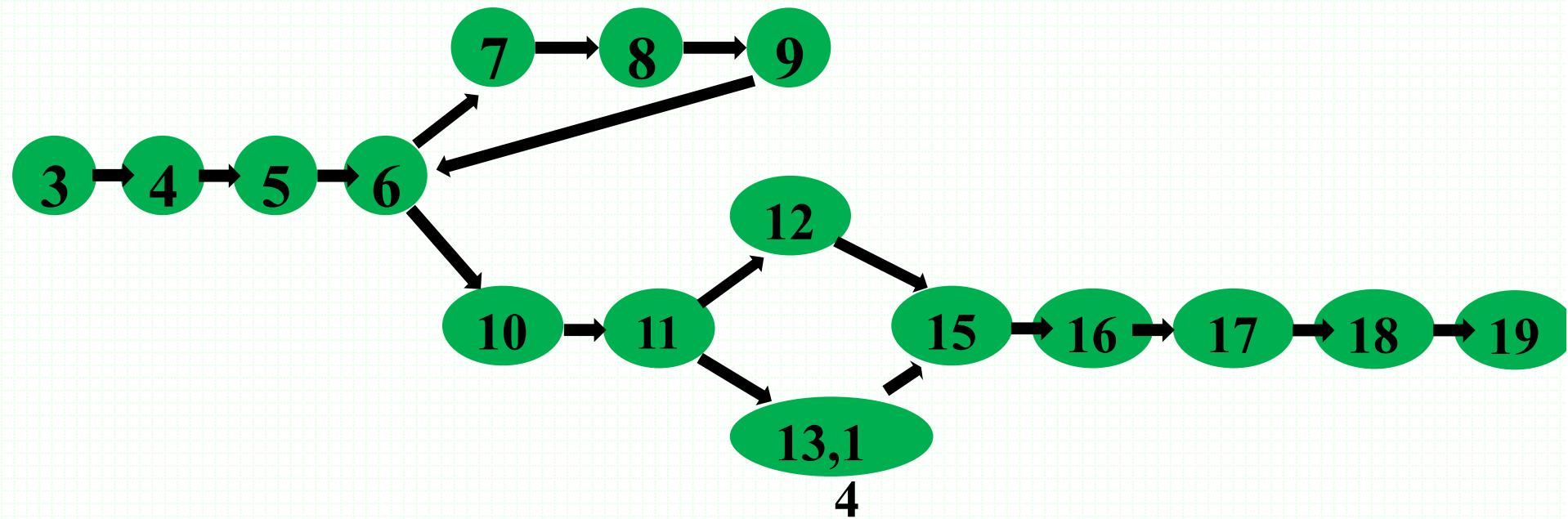
此路径为数据流测试路径，不同于前面的路径覆盖的路径



廖力

```
1 Program Example()
2 var staffDiscount, totalPrice, finalPrice, discount, price
3 staffDiscount = 0.1
4 totalPrice = 0
5 input(price)
6 while(price != -1) do
7   totalPrice = totalPrice + price
8   input(price)
9 od
10 print("Total price: " + totalPrice)
11 if(totalPrice > 15.00) then
12   discount = (staffDiscount * totalPrice) + 0.50
13 else
14   discount = staffDiscount * totalPrice
15 fi
16 print ("Discount: " + discount)
17 finalPrice = totalPrice - discount
18 print ("Final price: " + finalPrice)
19 endProgram
```

画出此程序的流图，  
写出其变量集合



程序流图



廖 力

## 动态白盒测试——数据流覆盖

### ■ 数据流覆盖基础

基本定义：

5. 定义节点**DEF(v,n)**——在节点n定义了变量v，  
即变量赋值语句

例如： **input x; x = 2;**

6. 使用节点**USE(v,n)**——在节点n使用了变量v

例如： **print x; a=2+x;**



廖 力

```
1 Program Example()
2 var staffDiscount, totalPrice, finalPrice, discount, price
3 staffDiscount = 0.1
4 totalPrice = 0
5 input(price)
6 while(price != -1) do
7   totalPrice = totalPrice + price
8   input(price)
9 od
10 print("Total price: " + totalPrice)
11 if(totalPrice > 15.00) then
12   discount = (staffDiscount * totalPrice) + 0.50
13 else
14   discount = staffDiscount * totalPrice
15 fi
16 print ("Discount: " + discount)
17 finalPrice = totalPrice - discount
18 print ("Final price: " + finalPrice)
19 endProgram
```

写出此程序中变量  
totalPrice的定义节  
点和使用节点

节点	类型	语句
4	D	<code>totalPrice = 0</code>
7	D	<code>totalPrice = totalPrice + price</code>
7	U	<code>totalPrice = totalPrice + price</code>
10	U	<code>print("Total price: " + totalPrice)</code>
11	U	<code>if(totalPrice&gt;15.00) then</code>
12	U	<code>discount = (staffDiscount * totalPrice) + 0.50</code>
14	U	<code>discount = staffDiscount * totalPrice</code>
17	U	<code>finalPrice = totalPrice - discount</code>



## 动态白盒测试——数据流覆盖

### ■ 数据流覆盖基础

基本定义：

7. 谓词使用P-use——USE(v, n)位于一个谓词中，即条件判断语句中

例如： if  $b > 6$

8. 运算使用C-use——USE(v, n)位于一个运算中，即计算表达式中

例如：  $x = 3 + b$



廖力

## 动态白盒测试——数据流覆盖

### ■ 数据流覆盖基础

基本定义：

C-use的进一步划分出：

9. 输出使用O-use——变量值被输出到屏幕/打印机
10. 定位使用L-use——变量值用于定位数组位置
11. 迭代使用I-use——变量值用于控制循环次数



廖 力

节点	类型	语句
7	CUse	<code>totalPrice = totalPrice + price</code>
10	OUse	<code>print("Total price: " + totalPrice)</code>
11	PUse	<code>if(totalPrice&gt;15.00) then</code>
12	CUse	<code>discount = (staffDiscount * totalPrice) + 0.50</code>
14	CUse	<code>discount = staffDiscount * totalPrice</code>
17	CUse	<code>finalPrice = totalPrice - discount</code>



## 动态白盒测试——数据流覆盖

### ■ 数据流覆盖基础

基本定义：

12. 变量v的定义-使用路径(**du-path**)：

给定PATH(P)中的某条路径，如果定义节点  
DEF(v, m)为该路径的起始节点，使用节点  
USE(v, n)为该路径的终止节点，则该路径是v的一  
条du-path。



廖力

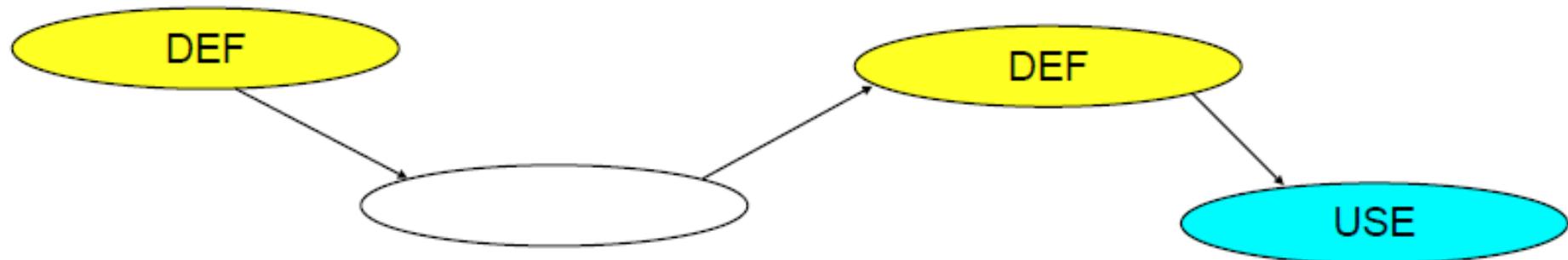


Figure 2: An example of a DU-path



廖力

```
1 Program Example()
2 var staffDiscount, totalPrice, finalPrice, discount, price
3 staffDiscount = 0.1
4 totalPrice = 0
5 input(price)
6 while(price != -1) do
7   totalPrice = totalPrice + price
8   input(price)
9 od
10 print("Total price: " + totalPrice)
11 if(totalPrice > 15.00) then
12   discount = (staffDiscount * totalPrice) + 0.50
13 else
14   discount = staffDiscount * totalPrice
15 fi
16 print ("Discount: " + discount)
17 finalPrice = totalPrice - discount
18 print ("Final price: " + finalPrice)
19 endProgram
```

写出变量totalPrice  
的du路径

p1: 4, 5, 6, 7  
p2: 4, 5, 6, 7, 8, 9,  
10 或 4, 5, 6, 10

## 动态白盒测试——数据流覆盖

### ■ 数据流覆盖基础

基本定义：

13. 变量v的**定义-清除路径(define-clear-path/dc-path)**:

如果变量v的某个定义-使用路径，除起始节点外没有其它定义节点，则该路径是变量v的定义-清除路径。



廖 力

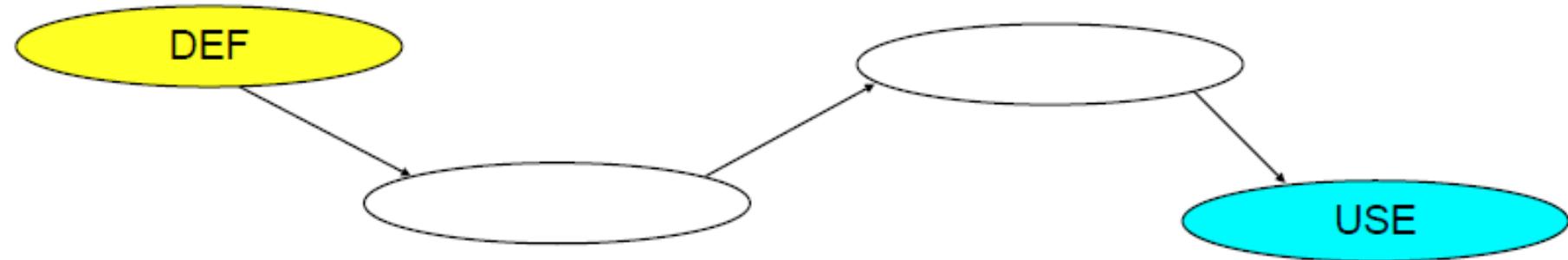


Figure 3: An example of a DU-path that is also definition-clear



廖力

```
1 Program Example()
2 var staffDiscount, totalPrice, finalPrice, discount, price
3 staffDiscount = 0.1
4 totalPrice = 0
5 input(price)
6 while(price != -1) do
7   totalPrice = totalPrice + price
8   input(price)
9 od
10 print("Total price: " + totalPrice)
11 if(totalPrice > 15.00) then
12   discount = (staffDiscount * totalPrice) + 0.50
13 else
14   discount = staffDiscount * totalPrice
15 fi
16 print ("Discount: " + discount)
17 finalPrice = totalPrice - discount
18 print ("Final price: " + finalPrice)
19 endProgram
```

写出变量totalPrice  
的dc路径?

p1: 4, 5, 6, 7  
p2: 4, 5, 6, 10

## 动态白盒测试——数据流覆盖

### ■ 数据流覆盖测试

*Step* 1. 对于给定的程序，构造相应的程序图

*Step* 2. 找出所有变量的定义-使用路径

*Step* 3. 考察测试用例对这些路径的覆盖程度，

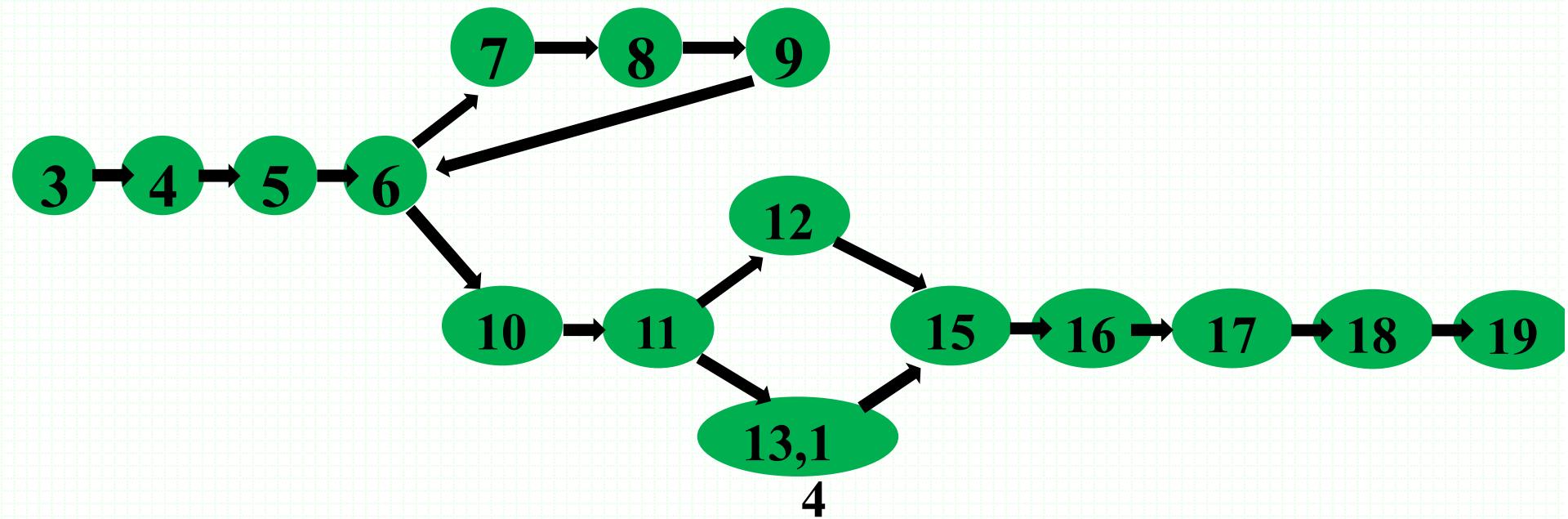
即可作为衡量测试效果的度量值



廖力

```
1 Program Example()
2 var staffDiscount, totalPrice, finalPrice, discount, price
3 staffDiscount = 0.1
4 totalPrice = 0
5 input(price)
6 while(price != -1) do
7   totalPrice = totalPrice + price
8   input(price)
9 od
10 print("Total price: " + totalPrice)
11 if(totalPrice > 15.00) then
12   discount = (staffDiscount * totalPrice) + 0.50
13 else
14   discount = staffDiscount * totalPrice
15 fi
16 print ("Discount: " + discount)
17 finalPrice = totalPrice - discount
18 print ("Final price: " + finalPrice)
19 endProgram
```

以变量totalPrice为例



程序流图



廖力

# 数据流测试练习——以变量**totalPrice**为例

## 1. 分析定义节点和使用节点

Node	Type	Code
4	DEF	<b>totalPrice</b> = 0
7	DEF	<b>totalPrice</b> = <b>totalPrice</b> + price
7	USE	<b>totalPrice</b> = <b>totalPrice</b> + price
10	USE	print("Total price: " + <b>totalPrice</b> )
11	USE	if( <b>totalPrice</b> > 15.00) then
12	USE	discount = (staffDiscount * <b>totalPrice</b> ) + 0.50
14	USE	discount = staffDiscount * <b>totalPrice</b>
17	USE	finalPrice = <b>totalPrice</b> - discount

1.1 统计变量出现次数

1.2 分析每次出现的节点类型

# 数据流测试练习——以变量totalPrice为例

## 2. 列举可能的DU路径

2.1 可能的路径数:  $2 \times 6 = 12$

2.2 列出DU路径

p1: 4, 5, 6, 7

p2: 4, 5, 6, 7, 8, 9, (6), 10 或 4, 5, 6, 10

p3: 4, 5, 6, 7, 8, 9, (6), 10, 11

p4: 4, 5, 6, 7, 8, 9, 10, 11, 12

p5: 4, 5, 6, 7, 8, 9, 10, 11, 13, 14

p6: 4, 5, 6, 7, 8, 9, 10, 11, 12, 15, 16, 17

p7: 7, 8, 9, (6), 7

p8: 7, 8, 9, 10

p9: 7, 8, 9, 10, 11

p10: 7, 8, 9, 10, 11, 12

p11: 7, 8, 9, 10, 11, 13, 14

p12: 7, 8, 9, 10, 11, 12, 15, 16, 17

# 数据流测试练习——以变量totalPrice为例

## 2.3 约简DU路径

p1: 4, 5, 6, 7

p2: 4, 5, 6, 7, 8, 9, 10

p3: 4, 5, 6, 7, 8, 9, 10, 11

p4: 4, 5, 6, 7, 8, 9, 10, 11, 12

p5: 4, 5, 6, 7, 8, 9, 10, 11, 13, 14

p6: 4, 5, 6, 7, 8, 9, 10, 11, 12, 15, 16, 17

p7: 7, 8, 9, 7

p8: 7, 8, 9, 10

p9: 7, 8, 9, 10, 11

p10: 7, 8, 9, 10, 11, 12

p11: 7, 8, 9, 10, 11, 13, 14

p12: 7, 8, 9, 10, 11, 12, 15, 16, 17

经过约简得到3条DU路径

# 数据流测试练习——以变量totalPrice为例

## 3. 设计测试用例

p5: 4, 5, 6, 7, 8, 9, 10, 11, 13, 14

price = 14, -1

p6: 4, 5, 6, 7, 8, 9, 10, 11, 12, 15, 16, 17

price = 16, -1

P7: 7, 8, 9, 7

Price=14, 12

```
1 Program Example()
2 var staffDiscount, totalPrice, finalPrice, discount, price
3 staffDiscount = 0.1
4 totalPrice = 0
5 input(price)
6 while(price != -1) do
7   totalPrice = totalPrice + price
8   input(price)
9 od
10 print("Total price: " + totalPrice)
11 if(totalPrice > 15.00) then
12   discount = (staffDiscount * totalPrice) + 0.50
13 else
14   discount = staffDiscount * totalPrice
15 fi
16 print ("Discount: " + discount)
17 finalPrice = totalPrice - discount
18 print ("Final price: " + finalPrice)
19 endProgram
```

以变量price为例

对于变量**price**,有两个定义节点和两个使用节点:

- Defining nodes:
  - DEF(price, 5)
  - DEF(price, 8)
- Usage nodes:
  - USE(price, 6)
  - USE(price, 7)



变量price对应4条du-path:

- <5, 6>
- <5, 6, 7>
- <8, 9, 6>
- <8, 9, 6, 7>

变量price对应的4条du-path同时也是dc-path



廖力

## 动态白盒测试——数据流覆盖

### ■ 数据流覆盖常用标准

- 1. Rapps 和 Weyuker 标准
- 2. Ntafos 标准
- 3. Ural 标准
- 4. Laski 和 Korel 标准



廖力

## 动态白盒测试——数据流覆盖

### ■ Rapps-Weyuker Metrics

S. Rapps and E. J. Weyuker, “Selecting software test data using data flow information.,” *IEEE Trans. Software Eng.*, vol. 11, no. 4, pp. 367–375, 1985.

#### 目标:

- (1) 保证所选路径数目总是有限的/可被实际处理的;
- (2) 寻找一种系统的测试路径选择方案，以帮助发现未知缺陷；



廖力

## 动态白盒测试——数据流覆盖

### ■ Rapps-Weyuker Metrics

#### 1. All-Paths:

等价于对流图的路径覆盖

#### 2. All-Edges:

等价于对流图的分支覆盖

#### 3. All-Nodes:

等价于对流图的语句覆盖



廖力

练习：对下列子程序进行数据流覆盖测试

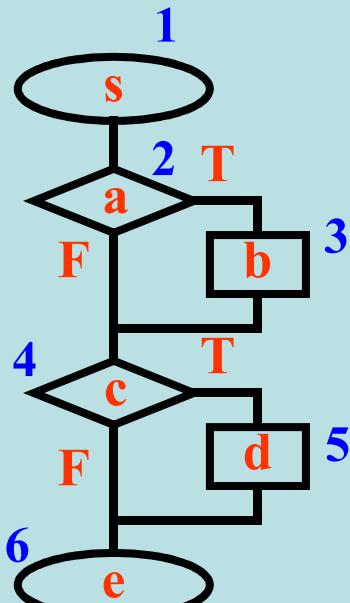
```
procedure example(y,z:real;var x:real);
begin
  if (y>1) and (z=0) then x:=x/y;
  if (y=2) or (x>1) then x:=x+1;
end;
```



廖力

# 数据流覆盖测试练习1

流程图



b:  $x = x/y$

d:  $x = x + 1$

2

2x

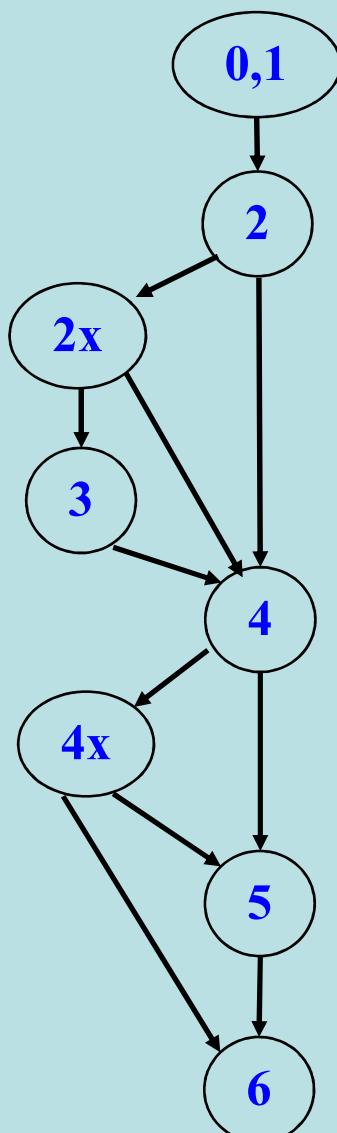
a:  $(y > 1) \text{ and } (z = 0)$

c:  $(y = 2) \text{ or } (x > 1)$

4

4x

流图



程序

```
0 procedure example(y,z:real;var x:real);
1 begin
2, 2x  if (y>1) and (z=0)
3  then x:=x/y;
4,4x  if (y=2) or (x>1)
5  then x:=x+1;
6 end;
```

## 动态白盒测试——数据流覆盖

### ■ Rapps-Weyuker Metrics

#### 4. All-Defs:

路径中的每个变量的每个定义节点都有一条dc-path到达该定义的某个使用节点

- The set of paths satisfies All-Defs for P if and only if, within the set of paths chosen, every defining node for each variable in the program has a definition-clear path to a usage node for the same variable, within the set of paths chosen.



廖力

## 动态白盒测试——数据流覆盖

### ■ Rapps-Weyuker Metrics

#### 5. All-P-Uses:

路径中的每个变量的每个定义节点都有一条dc-path到达该定义的每个P-use节点

- The set of paths satisfies All-P-Uses for P if and only if, within the set of paths chosen, every defining node for each variable in the program has a definition-clear path to every P-use node for the same variable.



廖力

## 动态白盒测试——数据流覆盖

### ■ Rapps-Weyuker Metrics

#### 6. All-P-Uses/Some C-Uses:

路径中的每个变量的每个定义节点都有一条dc-path到达该定义的每个P-use节点;但如果没有任何可达P-use节点, dc-path应到达至少一个C-use节点

- The set of paths satisfies All P-Uses/Some C-Uses for P if and only if, within the set of paths chosen, every defining node for each variable in the program has a definition-clear path to every P-use node for the same variable: however, if there are no reachable P-uses, the definition-clear path leads to at least one C-use of the variable.



## 动态白盒测试——数据流覆盖

### ■ Rapps-Weyuker Metrics

#### 7. All-C-Uses/Some P-Uses:

路径中的每个变量的每个定义节点都有一条dc-path到达该定义的每个C-use节点;但如果没有任何可达C-use节点, dc-path应到达至少一个P-use节点

- The set of paths satisfies All C-Uses/Some P-Uses for P if and only if, within the set of paths chosen, every defining node for each variable in the program has a definition-clear path to every C-use node for the same variable: however, if there are no reachable C-uses, the definition-clear path leads to at least one P-use of the variable.



廖力

## 动态白盒测试——数据流覆盖

### ■ Rapps-Weyuker Metrics

#### 8. All-Uses:

路径中的每个变量的每个定义节点都有一条dc-path到达该变量的每个使用节点,若存在定义节点和使用节点间存在多条dc-path,仅选取其中一条。

- The set of paths satisfies All-Uses for P if and only if, within the set of paths chosen, every defining node for each variable in the program has a definition-clear path to every usage node for the same variable.



廖力

## 动态白盒测试——数据流覆盖

### ■ Rapps-Weyuker Metrics

#### 9. All-DU-Paths:

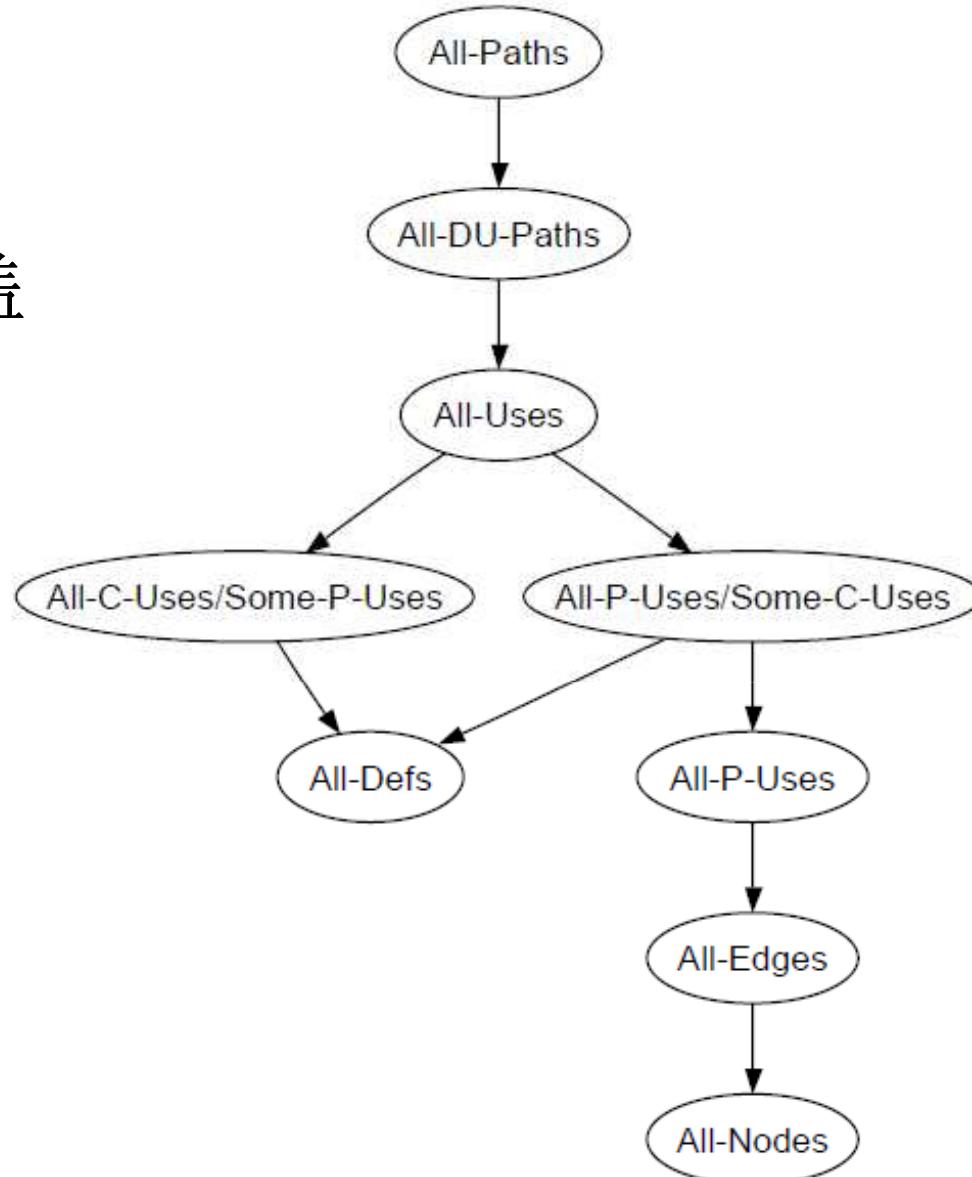
所有的**DU-Path**集合。路径中的每个变量的**每个定义节点**到达该变量的**每个使用节点**的所有**dc-path**。若存在循环，仅取一个简单循环或者无循环。

- The set of paths satisfies All-DU-Paths for P if and only if, the set of paths chosen contains every feasible DU-path for the program.



廖 力

## Rapps-Weyuker 覆盖 9条标准的关系



## 动态白盒测试

### ■ 优点

- (1) 检测代码中的判断和路径
- (2) 揭示隐藏在代码中的错误
- (3) 对代码的测试比较彻底

### ■ 缺点

- (1) 无法检测代码中**不可达路径**
- (2) 不验证需求规格



廖力

## 动态白盒测试VS静态白盒测试

	是否需要运行程序	执行自动化程度	测试结束条件	测试方法
动态	是	容易自动化	满足覆盖标准	形式化
静态	否	难以自动化	满足审查标准	低形式化



廖力

## 三 白盒测试工具



1. 主流的白盒测试工具有哪些？



廖力

## 白盒测试工具

### ■ 作用

提高代码分析效率，降低测试成本

### ■ 功能

静态分析工具

动态分析工具



廖力

## 白盒测试工具介绍

### ■ 主流白盒测试工具

-Rational 白盒测试工具集： LogiScope, Purify, ...

-ParaSoft 白盒测试工具集

-Compuware 白盒测试工具集

-XUnit系列工具： JUnit, CPPUnit, JTest, XMLUnit, ...

...



廖 力

## 白盒测试工具介绍

### ■ 静态分析工具

代表: Rational LogiScope

功能:

1. 在生命周期早期检测软件编码缺陷, 帮助缩短开发时间
2. 通过指出未测试的软件代码, 改进代码可靠性, 定位易出错的模块, 这些模块通常导致了大部分软件错误
3. 通过提供基于软件度量和图形的信息, 预测和诊断问题
4. 通过适当的代码优化重组, 识别源码树结构中重复的代码, 帮助降低维护成本

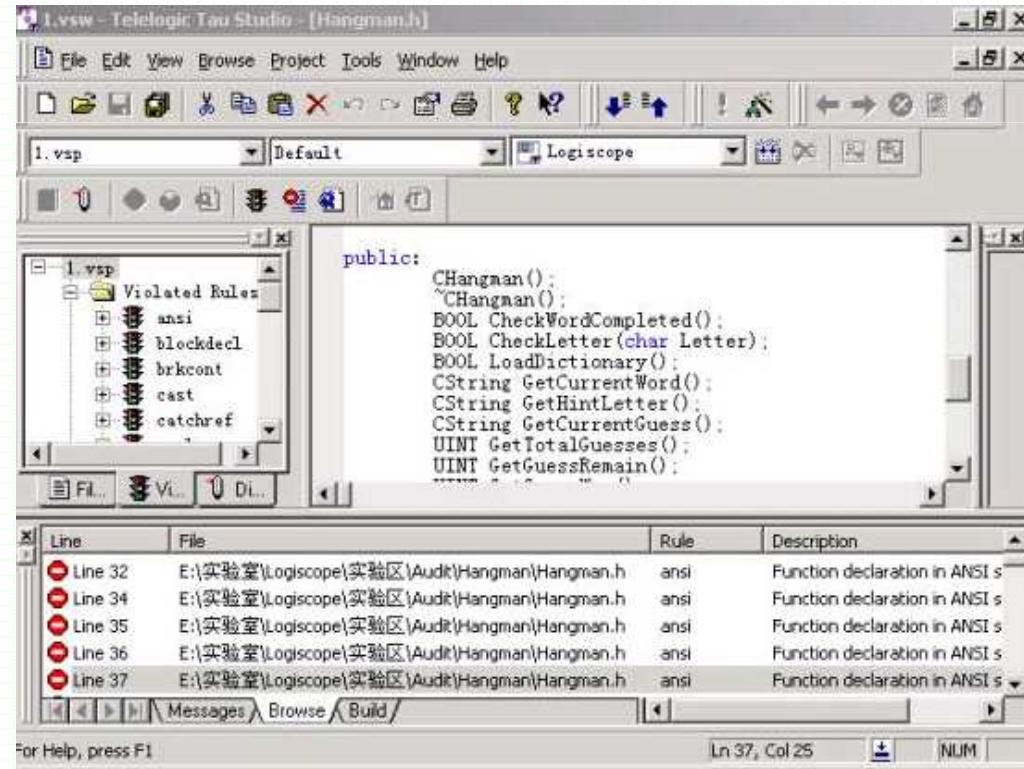


廖 力

## 白盒测试工具介绍

### ■ 静态分析工具

代表： Rational Logiscope



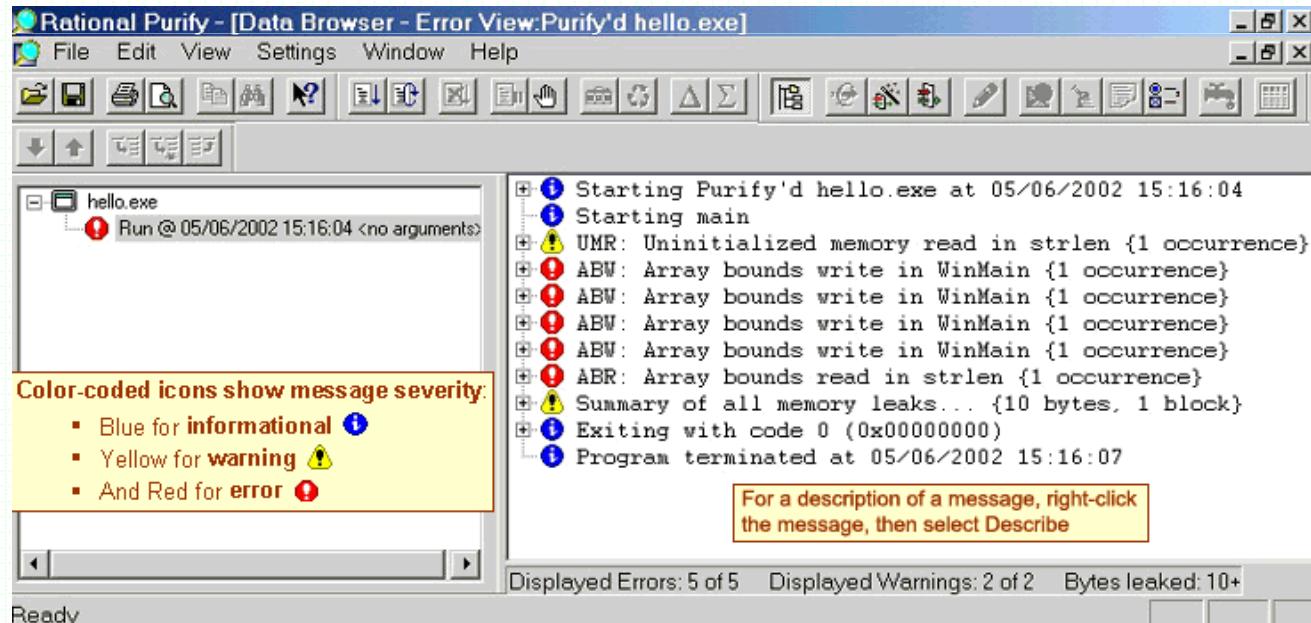
廖 力

## 白盒测试工具介绍

### ■ 动态错误检测工具

代表: Rational Purify

功能: 检查代码中类似内存泄漏、数组越界等错误



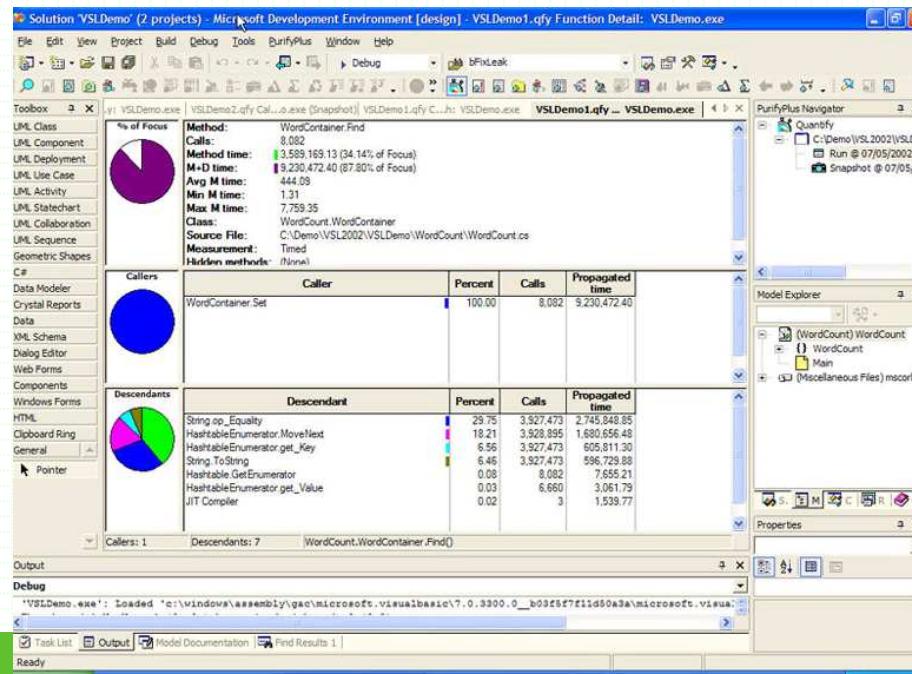
廖力

## 白盒测试工具介绍

### ■ 时间性能测试工具

代表: Rational Quantify

功能: 记录程序执行时间的细节, 包括语句或函数, 定位代码中的性能瓶颈

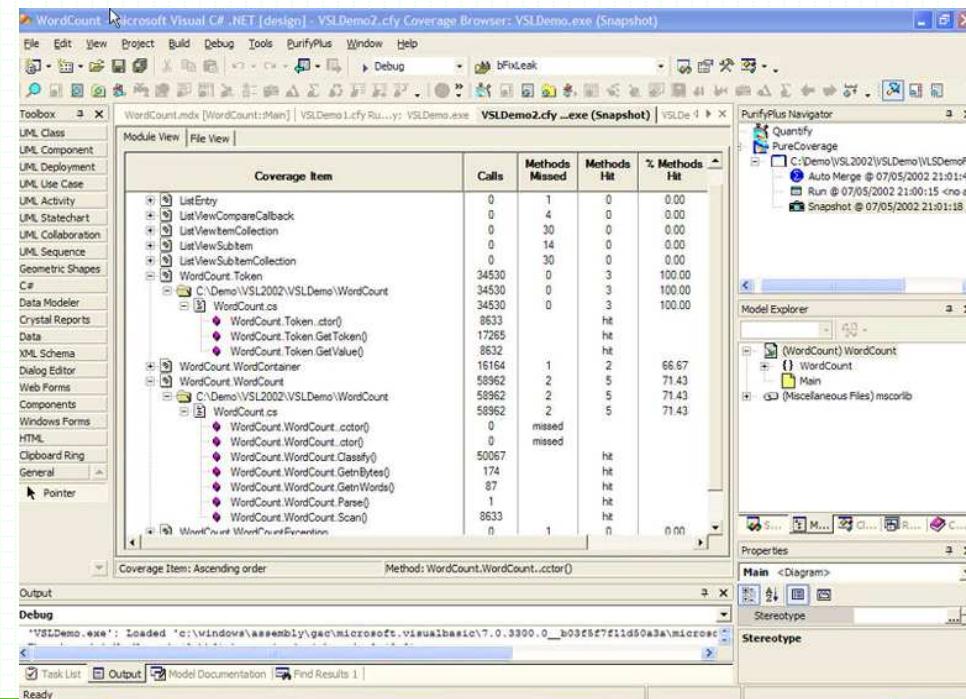


## 白盒测试工具介绍

### ■ 覆盖率统计工具

代表: Rational Coverage

功能: 统计当前测试用例对代码的覆盖率, 保证单元测试的全面性



## 白盒测试工具介绍

### ■ Rational 动态白盒测试工具原理

动态错误检测、性能分析和覆盖率统计的机理：用测试工具对被测程序进行编译、连接，生成可执行程序，该过程中，工具会对被测代码插桩，然后在后台收集程序的动态错误，执行时间和覆盖率信息；程序退出后，工具显示收集的数据



廖力

## 参考文献

1. 郁莲, 软件测试方法与实践, p18-p42, 清华大学出版社
2. 宫云战 主编, 软件测试教程, p50-p78, 机械工业出版社
3. Andreas Spillner, Tilo Linz, Hans Schaefer (刘琴 等 译), 软件测试基础教程, 人民邮电出版社
4. Srinivasan Desikan, Gopalaswamy Ramesh, (韩柯, 李娜 译), 软件测试: 原理与实践, p30-p46, 机械工业出版社
5. 《微软的软件测试之道》
6. Ron Patton, 《软件测试》
7. 《软件测试艺术》



廖力